The Automatic Implementation Of A Dynamic Load Balancing Strategy Within Structured Mesh Codes Generated Using A Parallelisation Tool

Jacqueline Nadine Rodrigues

A thesis submitted in partial fulfilment of the
requirement of the University of Greenwich
for the Degree of Doctor of Philosophy

March 2003

Parallel Processing Research Group
School of Computing and Mathematical Science
University of Greenwich
London, UK

I certify that this work has not been accepted in substance for any degree, and is not concurrently submitted for any degree other than that of Doctor of Philosophy (PhD) of the University of Greenwich. I also declare that this work is the result of my own investigations except where otherwise stated.

Acknowledgements

I would kindly like to thank several people for all of their help and support throughout this period of research.

In particular, I would like to take this opportunity to thank my supervisors, Dr Steve Johnson, Dr Cos Ierotheou and Professor Mark Cross, for their invaluable help and understanding they have given me.

I would also like to thank my colleagues Dr Peter Leggett, Dr Emyr Evans and Dr Chris Walshaw, who have assisted me in various degrees in the completion of this thesis.

I appreciate all the years of effort and skill that have gone into developing CAPTools, without which this research would probably not have been undertaken.

I would also like to acknowledge my sponsers: the EPSRC and the Univerisity of Greenwich.

Finally, I would like to thank my family and friends for the endless encouragement that they have shown me over the years.

In memory of TC.

Abstract

This research demonstrates that the automatic implementation of a dynamic load balancing (DLB) strategy within a parallel SPMD (single program multiple data) structured mesh application code is possible. It details how DLB can be effectively employed to reduce the level of load imbalance in a parallel system without expert knowledge of the application. Furnishing CAPTools (the Computer Aided Parallelisation Tools) with the additional functionality of DLB, a DLB parallel version of the serial Fortran 77 application code can be generated quickly and easily with the press of a few buttons, allowing the user to obtain results on various platforms rather than concentrate on implementing a DLB strategy within their code. Results show that the devised DLB strategy has successfully decreased idle time by locally increasing/decreasing processor workloads as and when required to suit the parallel application, utilising the available resources efficiently.

Several possible DLB strategies are examined with the understanding that it needs to be generic if it is to be automatically implemented within CAPTools and applied to a wide range of application codes. This research investigates the issues surrounding load imbalance, distinguishing between processor and physical imbalance in terms of the load redistribution of a parallel application executed on a homogeneous or heterogeneous system. Issues such as where to redistribute the workload, how often to redistribute, calculating and implementing the new distribution (deciding what data arrays to redistribute in the latter case), are all covered in detail, with many of these issues common to the automatic implementation of DLB for unstructured mesh application codes.

The devised DLB Staggered Limit Strategy discussed in this thesis offers flexibility as well as ease of implementation whilst minimising changes to the user's code. The generic utilities developed for this research are discussed along with their manual implementation upon which the automation algorithms are based, where these utilities are interchangeable with alternative methods if desired. This thesis aims to encourage the use of the DLB Staggered Limit Strategy since its benefits are evidently significant and are now easily achievable with its automatic implementation using CAPTools.

Contents

Figures

Chapter 1 Introduction

This Chapter aims to illustrate the need for dynamic load balancing (DLB) within parallel structured mesh codes. It gives an introduction to the reasons for parallelising an application code, along with various parallelisation techniques. An investigation into some of the reasons for parallel inefficiencies leads to the need for DLB, the motivation of this work. A summary of current DLB strategies is given in conjunction with several of the main issues relating to this area.

1.1 Introduction To The Problem

In the serial processing of Computational Fluid Dynamics (CFD) or Computational Mechanics (CM) codes (see Section 1.2), the speed and accuracy of the solution to a problem is fundamentally dependent upon how accurately the chemical and physical processes have been represented, and upon the geometrical accuracy and density of the mesh. In particular, more accuracy can often be achieved when refining the mesh density, which in turn takes longer to compute. A compromise between speed and accuracy is therefore often necessary, but parallel processing can be used to ease this problem such that several processors can undertake the work that was originally done by the single processor. The problem size is no longer constrained by the memory capacity of a single processor and so the user is able to achieve a higher degree of accuracy through the use of a finer mesh than was previously possible when using a single processor. Additionally, the problem size on each processor is essentially reduced, allowing the overall speed of computing to increase.

Ideally, the speed of processing should increase proportionally to the number of processors used, however this is usually not the case. Even if all of the processors had the same specifications (such as speed and workload), the overall execution time would still be affected by the parallel communications and other overheads, implying the need to investigate other reasons for parallel inefficiencies which this Chapter examines.

Weather prediction is an obvious example requiring large amounts of computer power. It is very difficult to predict the actions of a hurricane [1], as seen in 1992 when Hurricane Andrew hit the East Coast of the USA killing 26 people and costing $25 billion worth of damage [2]. If minimal damage is to be incurred then an early evacuation warning is vital to the residents living in the area in which the hurricane is expected to hit. Predictions need to be as accurate as possible in order for people to establish confidence in the warnings, otherwise there is a risk that future warnings will be ignored. This means that a large amount of data is needed to obtain the desired accuracy, which, if run in serial, may not be produced fast enough, leaving residents little or no time to evacuate or prepare for the oncoming severe weather condition. Using parallel processing means that the mesh density can be increased to obtain a higher level of accuracy, and then executed on a number of processors to produce information quickly in order to make a prediction. Examining the effects of severe weather conditions, such as the likes of El Niño and La Niña [3], can allow experts to forecast the foreseeable weather, and predict climate changes in hundreds of years time, enabling people to prepare for impending conditions.

The use of simulation models provide an important tool for solving many scientific problems which can be used to reproduce the results or behaviour of a certain event that would usually be either impractical or too expensive to perform experimentally. For example, with the introduction of The Comprehensive Nuclear Test-Ban Treaty [4], nuclear simulation is the only practical method of testing the nuclear stockpiles, where simulation is far cheaper and ethically sound than actual testing. Simulation allows the user to safely model a nuclear event without the need to deal with ethical issues or to use expensive equipment to measure extreme temperatures that may be physically impossible to monitor. Simulation enables the user to cost effectively perform numerous tests that could not have been performed manually due to practical constraints, such as the cost, safety and effort required to run the experiment several times.

For simulations to be useful they must be accurate, as it is sometimes impossible to actually compare results to the real-life observations. This is true in Metal Casting Models for example where the results are needed to detect faults in the casts, as the temperatures are so extreme that it would be impossible to perform any test on the true temperature without affecting the actual casting

model [5]. Consider simulations involving aircraft wings where it would be very expensive to physically perform the tests involved. Accuracy is needed to correctly simulate the tests in which the only alternative is to speculate. The simulation is used to detect possible faults (under certain conditions), where these can then be rectified before going into production, saving time, money, and lives. Accuracy is important if results are to be taken seriously, as the user needs reliability in order to make any informed decisions.

Speed is also an important issue in most fields, but particularly so because results may be needed quickly otherwise the results would become obsolete. For example, there would be no point in predicting the weather forecast for yesterday, as this would be useless to everyone, which suggests that the prediction is only valid if provided in time. Additionally, speed is important because of the costs associated with the time spent using the machine, which means that all calculations should be completed efficiently in order to limit the cost.

1.2 Structured Mesh Codes

Many problems being simulated can be modelled using either structured meshes or unstructured meshes, where the former is often used with finite difference techniques, and the later is often used with finite element analysis. Although the user is provided with a higher degree of geometric accuracy when using an irregular mesh (unstructured), the regular mesh (structured) offers simplicity and speed. Due to its flexibility, the user is capable of modelling more complex geometries when using an unstructured mesh, but this results in the need for indirect addressing which is slower than the direct addressing used with structured meshes. Although there are many benefits to using unstructured mesh codes, many codes are written using structured mesh codes because they are easier to code, and because computers are not capable of sustaining the speed required when using the alternative which was memory intensive. As a result, this project deals with structured mesh codes, the issues surrounding unstructured mesh codes are addressed at a later stage.

Using a structured mesh, the whole domain of the problem can be discretised, where calculations are performed on the mesh points or cell centres. For example, when predicting the weather, calculations can be performed on certain points across a discretised model of the globe, and similarly, a discretised model of a cast can be used to simulate the solidification process of molten metal poured into the cast. The shape of the mesh is dependent upon the geometry of the problem, where it is often necessary to use a rough fit in instances when a perfect fit cannot be made.

1.3 Serial Processing

In the past two decades the entertainment industry (including game developers for example) has been one of the key drivers to develop superior machines with more memory and faster processing power. Mathematical, chemical, and physical sciences all play a major role in the development of computational science, which aims to achieve far more than is currently possible. Many scientific application codes have been written over the past decades that aim to model, simulate, or solve, complex problems which cannot be solved efficiently by hand, since millions of calculations are needed to achieve a required degree of accuracy.

Most of the computationally intensive scientific application codes were written specifically for serial execution, as this was the only option available. The size of the mesh in the application code was usually dependent upon the memory capacity of the processor being used at the time the code was written, meaning that those groups with a lot of money were able to execute larger applications than other groups, as they could afford the superior machines.

1.4 Shared Memory Systems (SMS) And Distributed Memory Systems (DMS)

Either Shared Memory Systems (SMS), Distributed Memory Systems (DMS) [6] or a combination of both can be used in parallel processing, where this research is related to the use of the DMS. With SMS, the multiple processors operate independently but share the same memory resources. Only one processor can access a particular location of the shared memory at a time, where synchronisation is used to control processor reads and writes to the same location. With DMS, the multiple processors operate independently on their own private memory, where data is shared across a communication network by using message passing (which the user is responsible for synchronising).

1.5 Parallel Processing

The development of parallel processing was driven by the user's insatiable need for faster and more accurate results, as many CFD codes require a large amount of processing power. Using serial processing, these codes often take hours, or even days to run, implying the need to run these codes in a fraction of the time.

As technology progressed it became possible to use multiple processors concurrently to solve a problem rather than using just a single processor. Special parallel machines were developed which enabled the user to utilise the processing power of several processors together. Parallel processing allowed the user to improve the representation of the domains and also of the chemical and physical processes of their code, as the problem was no longer restricted by the time and memory capacity of the machine. These machines were expensive and therefore exclusive to those who could afford such a machine, limiting the growth of parallel computing.

The cost of parallel computers can often be prohibitive with at best only limited access available (due to the large number of users required to justify its purchase), however, it is no longer necessary to have access to a parallel machine

in order to run parallel code. Due to current advances in technology a cluster of workstations is now sufficient, making it possible for anyone to make use of the hardware that is already available.

Table 1.1 lists the advantages and disadvantages of using parallel processing over serial processing. The main reason for using parallel processing is that faster results can be obtained, which is the foremost reason for using a machine, as the problem is effectively shared. In addition, the problem size is no longer restricted to the memory capacity of a single processor but can now use the memory capacity of several processors. Similarly, the accuracy of the results obtained is improved when using parallel processing, as the mesh can be refined, or the physics can be improved. These three points make parallel processing very desirable, even offsetting the cost of running in parallel (which has decreased with the employment of workstation clusters).

The hardware for parallel processing is obviously available, however, the time taken to manually parallelise a code is a drawback that cannot be ignored. This task is prone to human error, where the parallel code may be written using either a new language, or by adapting existing sequential code to run on these machines (which can involve many man-months or years). Note that it is usually easier to maintain and optimise a serial code compared with a parallel code, and so the initial algorithm being parallelised should ideally be correct before parallelisation. The notion of 'processor communication' must also be considered, as this is an unfamiliar concept with serial processing. Additional costs are also associated with processor communication, discussed in Section 1.9.

Advantages:	Disadvantages:
Faster results – share the problem	Time taken to parallelise – write parallel code, or convert existing code (many man-months involved)
Increase mesh density - memory capacity of several processors	Cost of processors compared to the cost of a single processor
More accuracy – finer mesh or improved physics now possible	Maintenance and optimisation difficult – minor change to algorithm may require large change to parallel code
Can make use of available resources (cluster of workstations)	Additional costs – communication

Table 1.1: Advantages and disadvantages of using parallel processing as opposed to serial processing.

1.6 Goals Of Parallelisation

Figure 1.1 lists a number of goals that are used to parallelise a code, where different members of the parallel community place a varying degree of importance on each goal [7].

Changes to the serial algorithm should be avoided so that the parallel results are the same as the serial results (discounting the effects of round-off), providing the user with a degree of confidence that the parallel code is correct. Additionally, the parallel code should be recognisable, allowing the user to maintain and optimise their parallel code. The parallel code should be run in the same way as the serial code, where the only difference is a noticeable increase in speed of processing and the size of the problem that can be processed. The purpose of taking the time and effort to parallelise a code is wasted if this latter requirement is not met, as the user expects a significant improvement over serial processing. The final goal is used to ensure that the problem size is proportional to the total local memory size available on every DM processor.

1) Minimise changes to the serial algorithm
2) Recognisable code
3) Transparent parallel execution
4) Improve efficiency over serial processing
5) Efficient use of all available memory (only for DM)

Figure 1.1: Goals that are used to parallelise a code.

1.6.1 Challenges Involved In Parallelisation

A number of challenges exist, some of which are shown in Figure 1.2. It is important to ensure that minimal changes are made to the user's code, as this will enable the user to easily maintain and optimise their parallel code. If major changes are made to the original serial code then the user will be unable to recognise their code, which could lead to future problems when trying to maintain or optimise the code. Ideally, the user should be able to understand the parallel code without the need to know the exact details of the underlying operations.

If the parallel code is to be considered beneficial and worthwhile to the user then it needs to be efficient, which means that the user should be able to obtain accurate results quickly, as well as being able to run bigger problem sizes. If the user is prepared to invest in parallelising their code, then it is expected that the parallel performance will be a significant improvement over the serial performance. The user must also consider the cost of parallelising their code, such as the time and effort required by a user to actually parallelise the code, plus the cost of the machines being used.

Ideally, the parallel code should be generic, so that it can efficiently execute on any processor topology and on any hardware platform. The parallel code should be scalable, such that the user can execute the code on a number of different processor topologies without having to change the code. The user would like to be able to obtain speed-up relative to the number of processors, and so the parallel code should be written in a certain way to achieve this.

1) Maximise parallel efficiency
2) Parallel code should be scalable
3) Parallel code should be portable

Figure 1.2: Some of the challenges encountered when using parallel processing.

1.7 Parallelisation Techniques

Very often, it is the author of an application code that is given the task of parallelising it, which means either parallelising the code by hand, using a parallel compiler, adopting routines from a parallel library or using a parallelisation tool.

However, parallelisation techniques are also used on legacy code modernisation projects [8] for instance, where the application code will typically have been written by someone other than the person parallelising the code. In this instance, the paralleliser may not want to make many changes to the code as their understanding of it may be limited (in terms of the physics involved for example). They may not want to re-write part of the code to fit in with a particular parallelising library or environment, especially if this involves modifying the existing data structures to conform with using some type of template.

1.7.1 Manual Parallelisation

Manually parallelising an application code allows the user to have total control over the parallelisation, however, this can be a daunting task. As well as being prone to human error (due to the complexities involved) this task is mundane, as the same operation may be performed time and time again, especially when dealing with very large application codes in which the user may have to inspect tens of thousands of lines of code. One mistake or incorrect decision can have a devastating effect on the resultant parallel code, increasing the parallelisation time even further. The positive aspect of this approach is that it can lead to very efficient parallel code, since the user has spent a great deal of time and effort in the parallelisation.

1.7.2 Parallelising Compilers

Those users that opt to rely on parallelising compilers from a vendor usually anticipate their serial code to perform well through the insertion of 'directives', such as those used for OpenMP [9], together with a small amount of re-writing. Some satisfactory results have been produced for limited cases using a parallelising compiler for the shared memory system [10]. SUIF [11] and Polaris [12] are examples of parallelising compilers. The success of this approach relies on three key issues. The first issue is the level of sophistication of the compiler as the compiler has complete responsibility for the entire parallelisation, where any flaw in its thoroughness can be detrimental. The compiler should try and identify all of the data dependencies in the code, detecting the possible parallelism (see Section B.6). The second key issue relates to the strategic placement of parallelisation directives that take the form of structured comments (which are ignored by non-parallelising compilers). A high level of expertise is required in order to determine directives to either override data dependencies that the compiler failed to disprove, or to enforce certain data placement strategies. The final key issue relates to the user's ability to tune the application, as the parallel performance may not supersede the serial performance, in which case the user

must be prepared to iteratively inspect performance data and modify the program accordingly. In general, however, the production of a good parallel code relies heavily on the success of the parallelising compiler, where the user has little control over the parallelisation.

The promotion of High Performance Fortran (HPF) [13] has been widespread, however results for certain test cases have shown the parallel performance to be substandard [14]. The user is required to posses a significant amount of expertise when applying (with substantial effort) the HPF directives to their serial code. In the context of most dusty-deck Fortran codes, HPF is restrictive in that a great deal of re-writing and re-engineering is needed before the code is even suitable for HPF. For example, interprocedural mapping of arrays needs to be consistent, meaning that if a 2D array is passed into a routine then it should be treated as a 2D array inside the called routine and throughout the entire code.

1.7.3 Parallel Libraries

Libraries of parallelised algorithms exist such that an algorithm in the library can be used by different application codes. Instead of writing the algorithm, the user simply makes use of an existing algorithm, which in this case has already been parallelised. PETSc [15] and NAG [16] are example libraries that provide this service.

Although this option seems desirable, the user has to ensure that the parallel algorithm is compatible with their own code and will often have to write their application code to fit in with the data structures used in the library routines. Unfortunately not all applications will fit into these predefined computation models and templates the libraries offer. In such cases the parallel code may not even be implemented or will have to be executed at a reduced level of performance.

1.7.4 Parallelisation Tools

Parallelisation tools can be used to aide in the parallelisation of application codes. Tools such as Forge 90 [17], the Vienna Fortran Compilation System [18], D Systems [19], PARADIGM [20], ParaScope [21], KeLP [22] and the Computer Aided Parallelisation Tools (CAPTools) [23, 24, 25, 26], are all currently available or are being developed. Tools offer the user more control over the parallelisation of their application, often enabling a better visualisation of the code. Due to the interactive nature of the tools, the user is able to force sections of the code to be parallel, sometimes by transforming the code in some way using the tool. Additionally, this parallelisation technique is not as restrictive with the data structures used as with parallel libraries.

A brief comparison of some of the available approaches discussed here and in the previous Sections is given by Frumkin et al. [14], where it is evident that there is definitely a need for interactive parallelising tools to assist in the production of architecture-independent parallel codes. Although manually produced message-passing codes exhibit the highest performance (by applying user knowledge of the code and intended architecture), the time and effort required by the user is often significant. The user's effort can be reduced by shifting the machine-dependent implementation details to compiler writers and library builders with the use of libraries. If portability were not an issue, then machine-specific parallelising compilers, combined with detailed profiling and user tuning, would be capable of producing acceptable performance for small codes. With the great need to limit compile time, the thoroughness in which interprocedural dependence analysis could be applied is reduced, thus affecting the quality of the parallel code produced for complex applications.

1.8 Computer Aided Parallelisation Tools (CAPTools)

As this research was carried out at the University of Greenwich, the context of this research involves CAPTools, where an understanding of its philosophy and practicalities are discussed in Appendix A and Appendix B.

CAPTools is a semi-automatic parallelisation tool that can already be used to automatically generate a parallel F77 version of a given serial F77 code. The aim of CAPTools is to generate code that is as efficient as a code that has been parallelised manually, using a combination of parallel compiler technology and as much user interaction as is necessary. The criteria in Figure 1.3 are used to effectively parallelise industrial and scientific application codes onto massively parallel systems.

- *Handle real world Fortran application codes regardless of the perceived "quality" of those codes*
- *No allowance for performance limitations of the generated parallel code due to the use of automation*
- *Generate code that is recognisable to the user following well understood parallelisation techniques*
- *Generate code that is portable to as wide a range of parallel systems as is feasible*

Figure 1.3: Criteria used by CAPTools to effectively parallelise industrial and scientific application codes onto massively parallel systems.

CAPTools is targeted at facilitating the generation of parallel F77 code with standard DMS communication calls where the generated code is easily portable to any DMS. The parallel code that is generated by CAPTools adheres to the Single Program Multiple Data (SPMD) model [27] in which each processor executes the same code but on its own subset of the program data. The generated parallel code is as similar as possible to the original serial code, differing only in the insertion of communication calls and execution control masks that ensure each processor operates on its own data subset, allowing the user to easily maintain and optimise it.

The core success of CAPTools lies in its powerful symbolic, interprocedural, value based dependence analysis (Section B.6). User interaction is vital in trying to ensure an accurate dependence analysis, as the user is able to examine information provided by the system at any stage during the parallelisation, as well as provide additional information.

A partitioning strategy for a structured mesh code can be prescribed simply by defining a routine name, and a variable array name along with an index (or subset of that array), from which CAPTools will then use as a basis to produce a comprehensive decomposition of the mesh for all relevant arrays. The automatic

inheritance of partition information to all of the appropriate variables in all routines is applied, reducing the effort required by the user to partition the problem. The user can then use CAPTools to calculate and generate execution control masks that use the "owner computes" rule (Section B.8), followed by the calculation and generation of communication statements. CAPTools generates an execution control mask for every statement that requires one, where that statement executes only on the processors that own the partitioned data. The calculation and generation of communication statements involve the placement, merging, and generation of a minimum number of communications (to avoid high communication costs). Once this is complete the user is able to generate the parallel code.

1.9 Processor Communication

Although each processor usually only operates on its own workload (subsection of the original problem) it may often need to use data owned by a neighbouring processor, where this data shall often be referred to as the halo region (or overlap region). This halo data therefore needs to be transferred from the owning processor onto the requesting processor so that current and up-to-date values will be used, this can be achieved using communications calls. Communication calls are needed when processors do not own the current values of the data that they request, or when a global operation (such as a summation) is needed, or when handling I/O. Communications are placed within the code such that the communicated data is obtained before being used. This topic is covered in more detail in Section B.9.

Some form of inter-processor communication is necessary that will transfer data from one processor to another. This has its own costs attached to it in the form of communication latencies (startup costs), data transfer time, and scheduling issues. Too many communications, or very large amounts of data being transferred, can lead to a significant amount of communication time adding to the overall parallel execution time. Although the communication latencies can

affect the performance of the code, this inefficiency problem is really a hardware issue that can only be solved with an improvement in hardware technology.

Communications occur in certain places in the code, and a processor can typically only continue with its work once that communication has occurred. Although many hardware systems and algorithms take advantage of asynchronous communications (communications that execute whilst performing computations) [28, 29], numerous global synchronisation points usually exist in CFD codes such as at the end of a time step. Even with asynchronous communication, if one processor reaches the synchronisation point before the other processors then it shall have to wait for those processors to catch up before continuing.

1.10 Parallel Inefficiencies

There are several reasons behind parallel inefficiencies, such as the quality of the algorithms used in the code, the speed and memory capacity of the machines being used, and the distribution of data onto processors.

Although the user has total control over the quality of the algorithm being used in the code, the algorithm may not be suited to parallelisation. For example, rather than using an implicit solver involving many communications in parallel, an explicit solver could be used. The additional iterations needed to achieve the same accuracy as the implicit solver may still outweigh the cost of communicating a large amount of data every iteration.

The user may not have access to the fastest machines, and may have to settle for the available resources, which could mean that parallel efficiency is dependent upon the efficiency of the slowest machine. The overall execution time of a parallel run is equivalent to the time of the slowest processor, which means that the parallel performance will be affected even if just one processor is slow.

Poor parallel performance can also be the result of badly distributed data, such that the way in which the data is distributed across processors is causing inefficiencies. Each processor may physically receive an equal workload, but the computational workload may vary from processor to processor due to the nature of the problem being solved. As well as the processor specification, the geometry

and physical characteristics of the problem can also have an effect on the performance of the problem, where some processors have more work to compute upon than other processors (Section 1.11).

Several possible solutions for improving the parallel efficiency are available to the user. The user could try improving the algorithm, but this may not be a plausible solution if no alternative algorithm exists. A drastic option would be to rewrite the code perhaps using enhanced algorithms combined with better programming techniques, however, it may be very difficult for the user to undertake such a demanding task, and there are only so many improvements that can be made. Additionally, the user may not be able to identify the algorithm as the problem, due to issues surrounding the processor specifications and the nature of the problem.

The user could simply execute their parallel code on faster machines with a larger memory capacity. This option relies heavily on the premise that superior machines do exist, and that such resources are available to the user, offering no long-term solution to the problem of inefficiency (especially since the user will always want a faster machine to meet their growing needs).

The final option is to improve the distribution of data amongst the processors being used, where the varying processor specifications and the nature of the problem being solved are considered. Currently with CAPTools, the data is distributed without any regard for the processor specifications or the nature of the problem. It is feasible that these factors can be considered in the data distribution, suggesting that load balancing (redistribution) could be used as a method for improving parallel efficiency.

1.11 Load Imbalance

The parallel performance of an application code is mainly dependent upon the nature of the application code, upon the input data (e.g. the input data to forecast the weather over London would be different to the input data used to forecast the weather across the entire globe), and upon the hardware being used. It is unlikely that the user will be able to generically predict the events that occur within a given

application code, nor will they know the precise details of the code (such as the value of a particular variable that is used in a conditional statement). Additionally, it is unlikely that the user will have control over the number of users/jobs executing on a particular machine. Given a combination of these factors, it is unlikely that the user would be able to accurately load balance a given application code. If the user could load balance their code statically then the application would only be balanced for a specific hardware topology and only for a given set of input data, which is very restrictive.

The data is distributed fairly evenly with most parallelisations, where each processor gets an approximately equal amount of cells to process. An even distribution is used since this is the simplest method of distributing the workload, however, the parallel performance sometimes suffers simply because the correct processor load is not used. The initial distribution is often unsuitable as it is based on the assumption that each processor will have the same computational workload for the duration of execution, and that the processor speeds are the same. The previous Section hinted that the parallel performance is affected by the data distribution, which currently does not consider the processor specifications or the nature of the problem.

Although the obvious benefits of speed and accuracy are attainable due to parallel processing, there is a new issue that can dramatically degrade parallel performance, known as 'load imbalance'. The load is said to be imbalanced if there is a significant amount of idle time present in the system of processors. The following Sections look at the causes of load imbalance, classifying a few of the different types of load imbalance which should be treated differently (see Section 1.14). Note that in this thesis a code is said to be 'balanced' when there is no physical phenomena, such that each cell on every processor takes the same time to compute. A code is said to be 'imbalanced' if either the nature of the code involves some changing physical phenomena that affect the runtime, or the geometry of the problem is complex. For example, the Jacobi Iterative Solver (see Section 4.9.1) is a clear example of a balanced code, whereas a Metal Casting Model is an example of a code that can exhibit both a complex geometry and the presence of changing physical phenomena.

1.11.1 'Processor' Imbalance

'Processor' imbalance is the term used to describe the situation in which the variation between processors leads to parallel inefficiencies. This term is typically used when solving a balanced problem using a heterogeneous system of processors. In this situation each processor is given the same amount of workload (amount of cells to compute on), where every cell on a processor takes the same time to compute (computational load). Load imbalance occurs due to the variations between processor specifications, such as speed, memory capacity, and number of users or jobs, which the user has little control over. Even if just one processor is being heavily utilised, this will have a significant effect on the parallel performance of the code, as the overall execution time is limited by the time of the slowest processor. In this type of situation, processors can be referred to as being either relatively 'fast' or 'slow', since it is this component that defines a processor.

If all of the processors have the same specifications when solving a balanced problem (computing the same amount of work at the same rate), then they will all finish computing at the same time, utilising the available hardware efficiently. With processor imbalance, some processors are often faster than the others, meaning the faster processors are idle whilst waiting for the slower processors to finish computing. Consider the example in Figure 1.4, showing the processor times when computing 1000 iterations using a Jacobi Iterative Solver on a cluster of workstations. The execution time for this example is that of Processor 5, which is approximately 160 seconds, even though most of the processors finish computing within 40 seconds. The other processors are idle for approximately 120 seconds, which is not efficient usage of the available hardware.

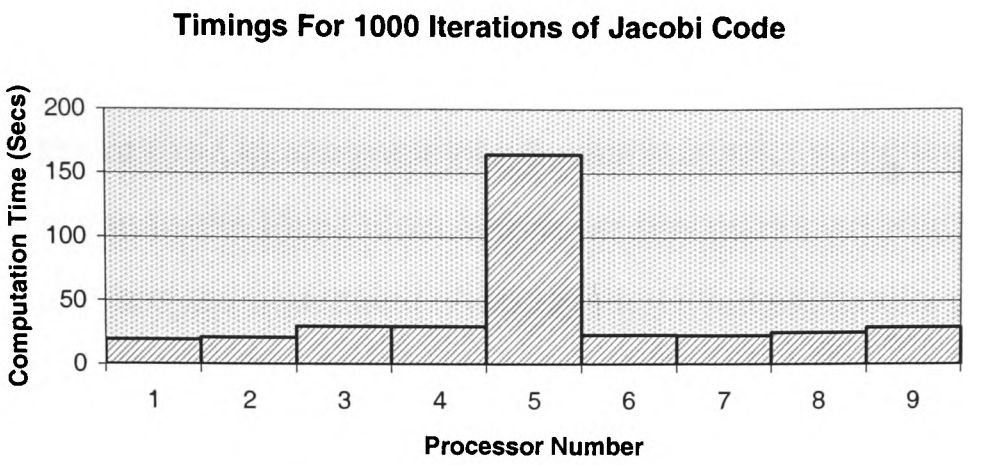


Figure 1.4: Example illustrating the difference between the processor timings for 1000 iterations of the Jacobi Iterative Solver used on a heterogeneous system of processors, where the overall time is that of the slowest processor.

1.11.2 'Physical' Imbalance

'Physical' imbalance is the term used to describe the situation in which the number of computations varies between processors, leading to parallel inefficiencies. This term is typically used when solving an imbalanced problem using a homogeneous system of processors, where the processor specifications are the same for each processor. The computational workload on each processor can vary either due to the geometry of the problem, or due to the physical characteristics of the problem, where many problems exhibit a combination of both. The processors can no longer be referred to in terms of 'fast' and 'slow' as each processor has the same speed, instead they are referred to in terms of 'heavily loaded' or 'lightly loaded'.

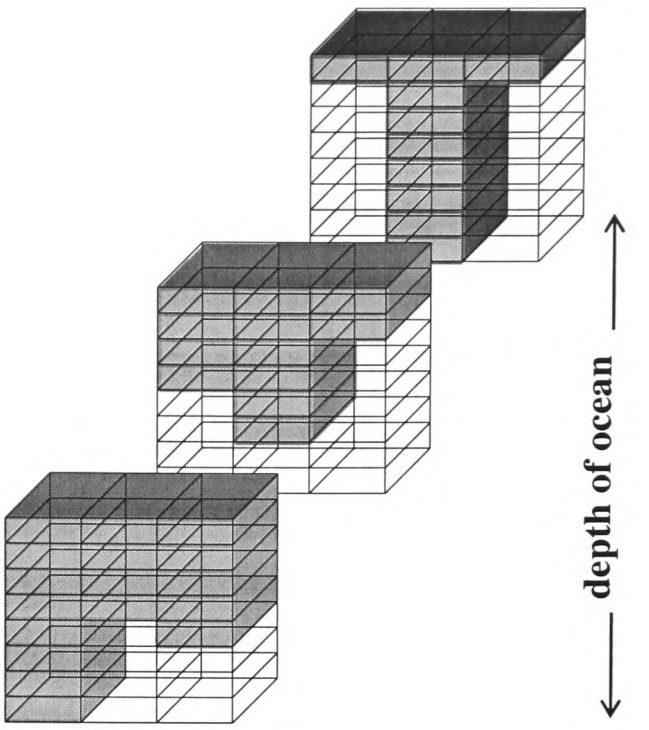
1.11.2.1 Geometry Of The Problem

An example area of science that exhibits geometrical imbalance is Oceanography and Climate studies (see Section 4.9.3). A suite of codes can be used, for instance, in weather forecasting, which is a good example showing the importance of obtaining fast and accurate results by means of parallelisation. Typically, parallelising these codes means that the discretised model of the Earth's surface is

partitioned onto a number of processors, each of which may own a number of land cells and a number of sea cells, as shown in Figure 1.5. The problem of parallel inefficiency arises in the Oceanography code, for example, when trying to model the flow of the ocean in the Fluid Flow Solver on processors owning land cells, as little or no calculations are performed. Although each homogeneous processor has a similar physical workload, calculations may only be performed on certain cells in that load depending on the geometry of the imbalanced code, where fluid flow calculations are only performed on the sea cells of each processor. This means that some processors will sit idle whilst waiting for other processors to complete their calculations, exhibiting natural imbalance. In Figure 1.5 for instance (using a system of homogeneous processors), Processor 1 (owning cells representing Europe and Russia) would have very little computational work in comparison to the middle processor computing flows for the Pacific Ocean. Processor 1 would be idle whilst waiting for the middle processor (represented by the black block containing most of the Pacific Ocean) to finish computing. Ideally each processor should have the same computational load to avoid the light processors remaining idle whilst waiting for the heavy processors to finish computing.

1	2	3
6	5	4
7	8	9

Surface Model (3x3)



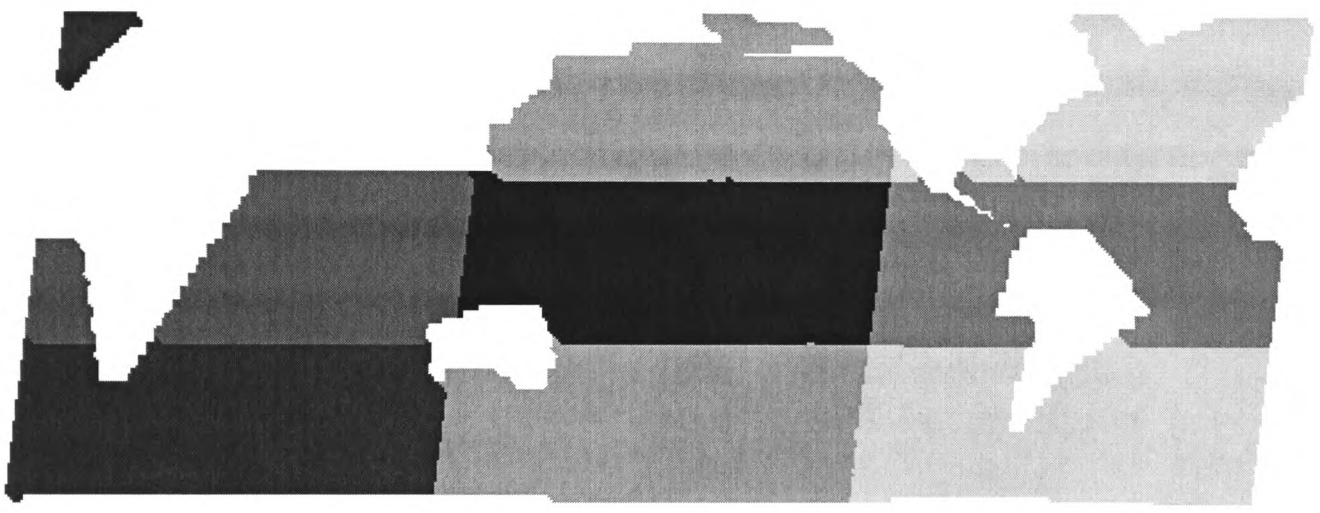


Figure 1.5: Example showing the Earth partitioned onto 9 processors, each represented by different colourings, where each processor owns a varying depth of ocean upon which to compute on. Africa and Europe are situated to the left and the Americas are situated to the right.

1.11.2.2 Physical Characteristics Of The Problem

Metal Casting is an example problem in which the physical characteristics of the code cause load imbalance. Monitoring this process is essential if faults are to be detected in the cast, as these can lead to further problems and prove costly [5]. The process of pouring molten metal into a cast and then cooling it down until solidified can only be simulated, as it is difficult to physically monitor the interior

of the casting where extreme temperatures are involved. The status of each cell in a mesh for a casting needs to be known at different stages in order to monitor the process, i.e. are cells either liquid (molten) or solid (solidified into the cast). This status then determines whether Fluid Flow or Stress/Strain calculations are relevant for a given cell.

Each processor initially has the same physical workload, where every cell on a processor shall be liquid, meaning that no calculations will be performed in the Stress Solver. As the problem solidifies (from the outside in), the number of solid cells owned by the boundary processors increase. In this example, the 'physical phenomena' refers to the solidification process, where the molten metal gradually solidifies across the processors. The load imbalance arises during the Fluid Flow Solvers where the processors containing mainly solid cells are idle, and vice versa for the processors owning mainly liquid cells during the stress solver. As an example, consider the following stages used when simulating the casting of a rectangular metal bar, which is cooled from one end through to the other in time:

1) Each processor owns all liquid cells (initial molten metal)
2) Some processors own a few solid cells but most own only liquid cells
3) A similar number of processors own all solid or all liquid cells
4) Some processors own a few liquid cells but most own only solid cells
5) Each processor owns all solid cells (solidified)

Figure 1.6 represents the casting of a rectangular bar at an intermediate stage in the solidification process. It should be noted that this example assumes that the bar is cooled from a particular end, and is used simply to demonstrate that some processors will have different amounts of work to compute depending on the status of their cells.

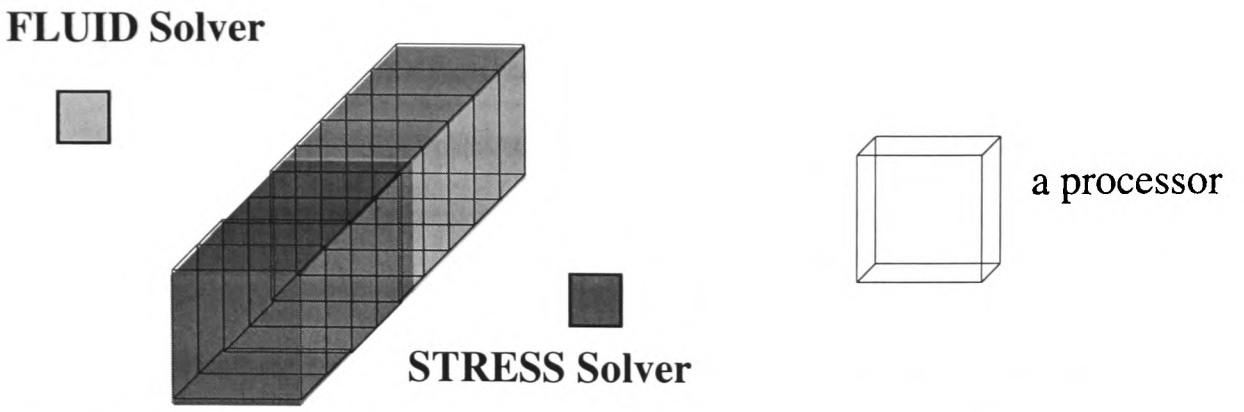


Figure 1.6: Example illustrating an intermediate stage in the solidification process of a rectangular bar, in which approximately half the cells are solid and half are liquid (for which different solvers are used), where the bar is cooled from one end.

1.11.2.3 Other Types Of Physically Imbalanced Problems

Adaptive mesh refinement problems [30] and crash impact problems [31] are examples that could also be classified as being physically load imbalanced. With parallel processing, some event (refinement/impact) in either case may lead to a single processor (or a few processors) having more computations to perform than the other processors.

1.12 Load Balancing

Although parallel computers are often used, their full potential cannot be realised unless larger systems are used that can be exploited with high parallel efficiency. The benefits from using massive parallelism are achieved for example by the UK Meteorology Office [32] using a CRAY T3E with hundreds of processors and by the Accelerated Strategic Computing Initiative (ASCI) [33] in the USA where thousands of processors are used to simulate nuclear explosions. Hundreds of processors are utilised to produce accurate results quickly. As the workload is processed at the speed of the 'slowest' or most 'heavily loaded' processor, there is potentially very poor efficiency in massively parallel systems. Idle time can occur on thousands of processors having a cumulative effect on parallel efficiency (for

example, with 1000 processors => idle time x 1000). This load imbalance must be reduced if it is going to be worth using parallelism on a large scale.

It has already been indicated that the initial distribution is not always the most practical, since various issues arise which cause this distribution to be unsuccessful, such as processor or physical load imbalance. The parallel performance of the code is dependent upon the configuration of the data across the processors, and so it would be beneficial to be able to redistribute the workload if necessary to improve the parallel efficiency. This Section aims to demonstrate the options available for overcoming the issue of load imbalance discussed in Section 1.11, where Static Load Balancing (SLB) and Dynamic Load Balancing (DLB), for structured mesh code problems are discussed in Section 1.12.3 and 1.12.4 respectively.

'Load balancing' is a term used to refer to the process of obtaining a balanced load. Rather than trying to improve the parallel efficiency by increasing the processor speed of the available machines, the workload is redistributed, offering a cheaper, long-term, solution to the problem of load imbalance (both processor and physical). Load balancing is becoming increasingly popular in the parallel community, where much effort has already been invested into improving parallel efficiency. Note that many of the current load balancing techniques relate to task balancing [34, 35, 36, 37, 38, 39], or the balancing of unstructured mesh based code [40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50], where the aim of load balancing is the same no matter what technique is used.

The purpose of any load balancing technique is to improve the parallel efficiency by decreasing the amount of idle time present in the system of processors, where the load is said to be imbalanced if either the processor speed, or computational workload, differs across the processors. The load is balanced such that the workload on the 'slowest', or most 'heavily' loaded, processor is reduced in order to curtail the overall execution time that is determined by this processor ('slow' or 'heavily' loaded). It is hoped that each processor will operate according to their capability and their defined workload, such that no processor is overloaded, with the aim that each processor will then finish computing at the same time.

With processor imbalance, for example, the load should be reduced on the slower processors in order to reduce the maximum processor timing (overall

parallel time). The load which is removed from the slower processors still needs to be processed, meaning that this load should be redistributed onto the other processors, preferably onto the faster processors rather than onto another slow processor (implying a load increase on the faster processors). This should result in a reduction of the idle time due to the fact that the slow processors now have less work to compute (reducing the maximum time), and that the faster processors now have more work to compute (increasing the minimum timing). This reduction of idle time leads to an improvement in efficiency. The same is true for physical imbalance in which the load is reduced on the heavily loaded processors, and increased on those processors with a light load. The cost of imbalance is essentially the time that can be saved (the difference between the maximum and 'average' timing), which suggests that the load should only be balanced if the redistribution cost is less than the cost of continued load imbalance. If the load is not redistributed then the load imbalance will continue and could even dominate the overall execution time.

1.12.1 Dynamic Scheduling On A SMS

With shared memory systems in which OpenMP directives have been inserted, it is possible to use the schedule clause to determine how iterations of a parallelised DO are split between the specified number of threads [9]. A chunk size can also be specified, indicating the number of contiguous iterations (iteration space) each thread will operate on. The default chunk size is 1 for dynamic scheduling and equal to the number of iterations divided by the number of threads for static scheduling. If the schedule is set to static then the iterations upon which a thread operates will not change during execution, whereas the opposite is true for a dynamic schedule where a thread will obtain the next set of iterations after processing its current iteration space. For example, when processing 14 iterations of a loop on 3 threads, then with static scheduling and a chunk size of 4 then thread 1 will process iterations 1 to 4 and iterations 13 and 14, thread 2 will process iterations 5 to 8 and thread 3 will process iterations 9 to 12. Similarly, if the same example was executed using dynamic scheduling and a chunk size of 2

then thread 1 will process iterations 1 and 2, thread 2 will process iterations 3 and 4 and thread 3 will process iterations 4 and 5. Iterations 6 and 7 will then be executed by the first thread that finishes processing their current iterations space, and likewise for the remaining iterations. This form of load balancing is not considered in this research simply because it is only applicable to applications executed on a SMS.

1.12.2 Task Balancing

With task balancing the tasks within the code are distributed between the processors on a first come first served basis. When a processor completes one task it is given another task by the master processor who is managing the system. There should be very little load imbalance with this method, although not all applications can implement this method.

Task balancing is not considered in this research because it involves excessive data movement as the entire mesh would need to be communicated every time a process finished its task. Additionally, this form of load balancing does not typically apply to most parallel structured mesh application codes executed on a DMS.

1.12.3 SPMD Static Load Balancing (SLB)

SLB refers to the situation in which the load is balanced just once (usually at the start of execution), using the same distribution throughout execution [41, 51, 52, 53] (compare with dynamic redistribution in Section 1.12.4). Essentially, a static partition is used in which the workload has been balanced using predictions of processor and/or physical imbalance. As stated earlier, with most parallelisations (including those performed using CAPTools) the data is distributed fairly evenly, with each processor getting an approximately equal amount of cells to process, however, the parallel performance can sometimes suffer simply because the

correct processor load is not used (as illustrated in Section 1.11). The idea behind SLB is that if the load were distributed differently to begin with then the parallel performance would not be so poor, since each processor would be operating on a suitably sized workload which they are capable of handling.

Table 1.2 offers some of the advantages and disadvantages of using SLB to improve the parallel efficiency of a code. The main benefit of using SLB is that it is very easy to implement, as the user need only calculate the workload just once for each processor, implying negligible changes to the user's code. This method of load balancing is suitable for handling problems with a static load imbalance, such as with 'geometrically' imbalanced problems. For instance, in the Oceanography example (Section 1.11.2.1), load imbalance was due to the complex geometry of the problem (whereby a homogeneous system of processors was used), where those processors owning mainly land cells were finishing before those processors owning mainly sea cells. This problem is said to be statically imbalanced because the number of land and sea cells did not change throughout execution, implying each iteration has the same amount of load imbalance. If the user could initially distribute the load so that each processor had roughly the same amount of sea cells, then the issue of load imbalance would not be as significant. The advantage of using SLB is that there is no need for further load balancing after using an initially balanced distribution, as the computational load remains constant.

The calculation that is used to obtain a balanced distribution is based on user knowledge of the problem, such as the geometry, the physics involved, and the processor specifications. For example, with the Oceanography problem, the user knew the general geometry of the problem was a map of the world, where the same computational load was associated with each sea cell, and that the processor specifications were the same (homogeneous system used). Even with this knowledge, however, the resultant balanced distribution would be based on an *estimate* of the static load imbalance, and not based on an accurate measure of the load imbalance. It is very difficult to accurately estimate the load imbalance in such situations, especially if the problems involve more complex geometries. Therefore, as well as requiring user knowledge of the imbalanced problem, it is difficult to use this knowledge to make an informed estimate of the load imbalance upon which the balanced distribution shall be calculated.

The main problem with SLB is that it is incapable of handling problems with continually varying load imbalance. Unlike the static load imbalance found with 'geometric' imbalance, the load imbalance can change continuously throughout execution. Consider the Jacobi problem (discussed in Section 1.11.1), for example, where a balanced problem is being solved on a heterogeneous system of processors. Although the variation between the processor speeds does not change, the number of jobs or users may change constantly during execution. Using SLB, the user may suggest a distribution based on the processor speeds, but this distribution may not be suitable due to the external factors mentioned. Additionally, this raised the question of whether this distribution would still be suitable if one of the processors were replaced by a completely different processor. The user has little chance of knowing exactly how many jobs or users will be running on a particular processor at any given moment, and so it would be impossible for them to estimate the balanced distribution.

The effects of the external factors are highlighted when examining the use of SLB with physically imbalanced problems. As with the processor imbalanced problem, the load imbalance of a physically imbalanced problem can change continuously throughout execution. The difference between a physically imbalanced problem and a processor imbalanced problem is that the user has no knowledge of the physical characteristics of the problem at any given time. With the processor imbalanced problem, it was possible for the user to make an estimate of the load imbalance since it was known that there were no variations due to the geometry or computational load of the problem. However, with physical imbalance on a homogeneous system of processors where the problem is geometrically balanced the computational load (due to physical phenomena) is the unknown varying factor. Consider the Casting problem (discussed in Section 1.11.2.2), where the load is initially balanced, since all of the processors contain liquid cells (the molten metal). An estimate of the load imbalance, based on the initial conditions of the problem, may indicate that there is no need for SLB, as the load is already balanced. The same is true if the estimate were based on the final conditions of the problem (where all of the cells have solidified). It is obvious from this example that redistributing the load once will not be sufficient, as the effects of load imbalance would only be delayed and not reduced.

Advantages:	Disadvantages:
Only calculate distribution once at start – no load migration during execution	Need knowledge of code to make decision
Negligible changes to the user's code	Difficult to accurately estimate workload
Suitable for constant (static) variation	Not suitable for continuous (dynamic) variation

Table 1.2: Advantages and disadvantages of using Static Load Balancing.

The points raised in this Section emphasise the fact that SLB is not suitable for handling both processor and physical imbalance (even though it could be used for 'geometrically' imbalanced problems). Even though SLB is easy to implement, the difficulty in accurately estimating a balanced distribution and the need for user knowledge of the code execution (processors, geometry, and physics), make this a poor solution to overcoming the effects of load imbalance in the general case.

1.12.4 SPMD Dynamic Load Balancing (DLB)

DLB refers to the situation in which the load distribution can be balanced several times during execution [54, 55, 56, 57, 58]. One advantage of using DLB as opposed to SLB is that the load can be balanced whenever required, which would not be possible with SLB if the load only became imbalanced half way through execution.

Table 1.3 offers some of the advantages and disadvantages of using DLB to improve the parallel efficiency of a code, which can be directly compared against those given for SLB. The main advantage of using DLB over SLB is that DLB is capable of handling problems with dynamic load imbalance (seen in processor and physically imbalanced problems), as well as static load imbalance ('geometrical' imbalance). As the processor specifications change (as in processor imbalance), or as the physical characteristics of the problem change (physical imbalance), the level of load imbalance is measured. This runtime measurement of the level of load imbalance can then be used to obtain a balanced distribution, accurately dealing with the continually changing load imbalance (compare with

Section 1.12.3). The main benefit of using a runtime redistribution is that it enables the problem to be balanced based on the current level of load imbalance rather than a particular estimate of load imbalance, which occurs with SLB. Additionally, there is no need for any user knowledge of the code specifics, such as the processor specifications, the geometry of the problem, or the varying physics of the problem (which is unknown), as this is incorporated into the measurement.

Consider the Casting problem again, containing physical imbalance, where the physical characteristics of the problem are changing throughout execution. As well as requiring a different distribution for each of the various stages of solidification (Section 1.11.2.2), several distributions may be necessary in order to achieve a satisfactory load balance. Similarly, DLB can also be used to handle 'geometrical' imbalance, where the load need only be balanced once (or more, if required), the advantage being that the balanced distribution is based on a measure of the load imbalance rather than being based on an initial estimate (which is less accurate).

There are several drawbacks associated with using DLB, most of which relate to the costs of balancing with this method. Although there is a cost related to calculating the distribution several times (that is only performed once with SLB), it can be argued that this is a minor cost compared to the cost of the continuously changing load imbalance. DLB also requires the load to be migrated to ensure processor ownership of data (Section 1.14), incurring additional costs and making changes to the user's code inevitable. However, the overall benefits offered by DLB make this the suitable method of load balancing.

Advantages:	Disadvantages:
Suitable for continuous (dynamic) variation	Calculate distribution several times
Accurate runtime measure of load imbalance	Load migration necessary (additional costs)
No need for knowledge of code	Additional changes to the user's code
Suitable for constant (static) variation	

Table 1.3: Advantages and disadvantages of using Dynamic Load Balancing.

1.13 Motivation For Research

It has been seen that parallel inefficiencies can arise from certain factors that are not under the user's control (Section 1.11). As the need for parallel processing is increasing, the use of load balancing techniques for combating parallel inefficiencies is becoming popular. It has been established that many structured mesh application codes exhibit parallel inefficiencies, where one of the main causes for parallel inefficiency is the effect of load imbalance. Different classifications of load imbalance were defined, where an application was said to contain either processor or physical imbalance, or a combination of both. In either classification, some processors would remain idle whilst waiting for other processors to finish computing, implying the inefficient use of the available hardware (since all of the processors were not continually busy throughout execution).

The differences between Static and Dynamic load balancing were examined in Section 1.12, where DLB showed evidence of attaining a better quality of load balance. More importantly, DLB shows evidence of being able to cope with both processor and physical imbalance, in which the load imbalance is changing continuously throughout execution.

The aim of DLB is to improve the parallel performance of the application code in question. This does not necessarily mean that the 'optimal' performance (load balance) will be obtained, but that the 'worst' case scenario will be greatly improved upon. It is unlikely that the 'optimal' performance could be obtained, as it is would be very difficult to predict the load balance. For example, the load may change continuously, or another user may log on to one of the machines being used. Ideally a generic DLB strategy that can be automatically implemented within a parallel SPMD code should be developed so that it can be applied to a wide range of application codes, allowing the user to obtain results in a smaller time frame.

1.14 Current Strategies And Issues Relating To Dynamic Load Balancing

More people use serial processing rather than parallel processing in the world simply because it is easier to code and requires less expertise. Out of those who use parallel processing only a number use the message passing paradigm. More importantly, out of those who use parallel processing with message passing, even fewer people use dynamic load balancing, as a tremendous amount of effort is required to implement dynamic load balancing within a parallel code.

Several issues relate to DLB, shown in Figure 1.7, all of which must be addressed [59 and 60]. Note that most of the issues discussed in this Section do not apply to SLB. With SLB the load is redistributed just once at the beginning of execution and so there is no need to even change the distribution. The only common issue with DLB is that of calculating the distribution to be implemented.

The importance of correctly identifying the section of the application code containing the load imbalance is discussed in Section 1.14.1, where this stage never even has to be considered with SLB. The decision of how often to redistribute the workload is discussed in Section 1.14.2, emphasising that the frequency of redistribution will usually be different for every application depending on the type of load imbalance. Sections 1.14.3 and 1.14.4 deal with the calculation and implementation of the new distribution respectively, where each relates to the other.

- *Where to redistribute in the parallel code*
- *When to redistribute (how often)*
- *Calculate the new distribution (partition)*
- *Implement the new distribution (move all necessary data)*

Figure 1.7: Issues relating to the implementation of Dynamic Load Balancing.

Apart from when calculating the new distribution, all of the other issues mentioned will be different for every application code, where the implementation of the new distribution is the most difficult to deal with. User decisions, or existing algorithms, can be used with the other issues, whereas the implementation is usually strongly related to the application code itself.

Several of these issues are also applicable to other load balancing techniques, although very little of the current research addresses all of the issues mentioned in Figure 1.7. For example, with dynamic load balancing for unstructured mesh application codes, single cells can be moved, making this option more flexible than dynamic load balancing for structured mesh codes. Graph partitioning tools such as Jostle [61, 62, 63, 64, 65] and Metis [66, 67] are used to determine the new distribution, but the other issues remain the same.

1.14.1 Where To Redistribute The Workload

DLB allows the distribution to be changed several times during the parallel execution of the code. The user has a choice regarding the location of redistribution, where the load may be redistributed at any location within the code. Improvements in parallel performance due to DLB are dependent on the location of redistribution, implying the importance of correctly identifying load imbalance. User understanding of the code is often required, where user knowledge or a profiler can be used to identify the load imbalance. Most load imbalance occurs within loops (such as time-step, iteration and solver loops), where large amounts of computation are being performed on every processor. In terms of granularity, redistribution would need to be very cheap if redistributing at the inner loop level (solver loops), as it would be performed many times. The amount of work (computations) between iterations of the loop would need to be considered if redistributing at the outer loop level (time-step) since the level of imbalance may become very significant before the next iteration is reached.

This stage can be time consuming, particularly if the user is not familiar with the code that is being balanced. Many of the current DLB strategies do not comment on any possible locations at which to redistribute the workload, although Cermele et al. [68] state that they leave this decision solely to the user.

1.14.2 Frequency Of When To Redistribute The Workload

The idea of balancing the workload distribution is to reduce the idle time present in the system of processors, achieved by reducing the maximum processor time. Considering the cost of redistribution, the question of how often to redistribute is an important issue with DLB. If the load is not balanced frequently enough (hardly ever balanced), then a significant amount of idle time will continue throughout execution, whereas the redistribution time will dominate the overall execution time if the load is balanced too often or every iteration (unless of course the redistribution time is free). Some of the current strategies have designed their own tests which indicate when to redistribute the workload [37, 68, 69, 70], some of which are based on a set number of loop iterations [71, 72, 73, 74, 75, 76, 77], or triggered when the timed proportion of imbalance exceeds some threshold [77, 78], or are based on either Unix calls [70, 72, 79] or micro-benchmarks [77] that measure the processor speed at the start of the run. With the latter case, the issue of physical imbalance was not considered, as the measurement of load imbalance was based on the variations between processors (timers were not placed around the imbalanced code). The problem with some of the current methods is that the user is expected to produce certain performance measures, such as the expected level of load imbalance, or how often the load should be redistributed (activated at the end of fixed intervals or phases) [77].

Timers can be placed around the imbalanced code, where a runtime measurement of the load imbalance is obtained, which considers both processor and physical imbalance. For example, Garner et al., who implemented DLB within the CAVITY code, invoke load balancing whenever any of the processor timings of every five iterations differ by more than 10% from the average timing, allowing rapid adjustment to varying conditions during a long run [80].

User familiarity with an application code often leads to specific DLB techniques, most of which may not be applicable to a wide range of applications, suggesting the need to find algorithms and performance metrics that are generic and not specific to the code itself.

1.14.3 Calculating The New Partition

This issue deals with the actual calculation of the new distribution with consideration for processor and physical imbalance. A new distribution (partition) needs to be determined based on the current partition, and on the current level of load imbalance. The current level of load imbalance is used as this gives the current status of the application code at a particular moment in execution. To avoid changing the distribution completely, which could involve a significant amount of data transfer (Section 1.14.4), the new partition should be based upon the current partition, where data is only transferred between neighbours in any given redistribution. If it was possible to completely change the distribution, then this could lead to significant redistribution overheads (particularly considering that many variables may need to be moved).

The granularity of the structured mesh code has to be considered, as it is not desirable to move single cells as can be done with DLB for unstructured mesh codes using the likes of Jostle [65] and Metis [67] (see Chapter 6), as this would involve many changes in the code. With DLB for structured mesh codes, only an entire row (or column, or plane, etc) of cells may be moved (see Section 2.2), implying that an optimal load balance may never be attainable. The basis behind this research is that the DLB algorithm should be cheap to perform if it is to be used dynamically. Several of the current methods of calculating the new workload make use of some sort of load balancing system [69, 70, 78, 79, 81] such as DAME [82, 83], where only Baillie et al. [84] acknowledge the influence of physical imbalance. They tend to require the user to implement their application code using the data structures of the load balancing system, which does not allow the user to easily change an existing code.

1.14.4 Implementing The New Partition

Another major issue with DLB concerns the implementation of the new distribution. The term 'load migration' is sometimes used to describe the process of implementing the new distribution, since some of the load is migrated onto the

new owners of the data. Load migration involves communicating the data between the processors to ensure processor ownership of the new partition. Migration is essential to the correctness of the parallel code execution, as the processors need to own the data that they operate on. The parallel code will not execute properly if even one item of data is not transferred correctly to the owning processor.

Some of the current DLB strategies use restart files to implement the new distribution [45, 85], where the details relating to the new partition are stored in a file after which execution is terminated. The parallel code is then executed again, this time loading in the restart file containing information about the new partition.

Other DLB strategies make use of a DLB system, such as DAME [82] and PLUM [48] for instance, in which the application code is explicitly written using the data structures needed for the load balancing system. DAME provides support for hiding irregular network topology, managing irregular data distribution and masking dynamic modifications of processor computational power. Current documentation suggests that it handles processor imbalance, but makes no mention of physical imbalance. It examines the state of the network and computational power of each processor at compile time, as well as performs runtime monitoring support where transparent checks are made at regular intervals. DAME automatically activates a mechanism that provides data migration from overloaded to underloaded processors. Program execution is interrupted, information is collected and then a decision is made about redistribution. PLUM is an automatic and architecture-independent framework for adaptive numerical computations in a message passing environment which is capable of handling problems with evolving physical features. It consists of a partitioner and a remapper that load balance and redistribute the computational mesh when necessary.

1.14.5 Manual Implementation Vs. Automatic Implementation

Manually implementing a DLB strategy is complicated and prone to errors, and can be difficult to maintain and optimise. Ensuring the correctness of the

implementation can be difficult too, since it is very likely that the user will make a mistake. It may be difficult to ensure that every stage of DLB is complete, where it may be necessary to reiterate some of the stages several times (especially when trying to implement the new distribution correctly). Implementation can also be affected by the size of the code, where the user may not be able to work on the entire code in one instance (due to visual limitations). Automation of a DLB technique avoids these problems and can encourage the far wider use of DLB, enabling more users to make efficient use of parallel hardware.

This Chapter has introduced DLB as a way of combating the effect of load imbalance, so its automatic implementation will significantly reduce the effort required by the user, leaving them free to obtain results. Although much research has been done and is still ongoing in many aspects of DLB, none of them encompass all phases of DLB in a generic sense, and do not provide the complete route towards automation adhering to the requirements of CAPTools parallel code. A generic strategy that uses many of the ideas from previous research is needed before automation can be realised.

1.15 Aims Of This Research

The four key aims of this research are shown in Figure 1.8. The main aim of this research is to devise a generic DLB strategy that will improve the parallel performance of a structured mesh application code, the fundamentals of which were discussed in Section 1.14. Whether a 2D or 3D (etc) partition has been employed, the DLB strategy should work correctly. It is hoped that the strategy would be automated within CAPTools, increasing the functionality of CAPTools with obvious benefits to the user. The difficulties associated with implementing DLB should be reduced when using CAPTools to automatically implement DLB within the user's code. The strategy needs to be generic if it is to become a practical feature within CAPTools, enabling DLB to be applied to a wide range of codes (rather than applicable to a specific application problem). Additionally, testing of the implemented DLB strategy is easier with automation, as focus can then be placed on the strategy, such as tuning some of the algorithms used, rather

than on the task of implementation itself. Note that the automatic implementation of the DLB strategy is only possible if its manual implementation proves feasible.

The devised DLB algorithm should be independent of the type of parallel machine being used, independent of any application code, and also independent of the input data of that application code. This allows any parallel machine, and any application code with any input data, to exploit the generated DLB parallel code. Only the implementation details of the generated code should be totally application dependent (for example, which arrays need to be migrated).

The purpose of dynamic load balancing has been demonstrated by the examples given earlier, where speed and accuracy are of great importance. To improve the efficiency of an imbalanced code the idle time must be reduced, which can only be achieved by redistributing the load when the level of load imbalance becomes significant, enabling processors to finish computing in the same amount of time. A major cost of a DLB algorithm is the time to calculate the new distribution and redistribute the program data, especially since a large number of program arrays may need to be migrated to satisfy a new partition. If this migration is too expensive, the improvements achieved by the new partition may be offset by the redistribution cost. An algorithm is required where the profitability of load redistribution is measured by taking both the level of idle time and the cost of redistribution into account and only applying it if profitable.

Additionally, the user should still be able to recognise and maintain the DLB version of the application code to allow continued maintenance and optimisation, and so utilities should be developed to avoid major changes to the code. To maximise the effectiveness of this, the cost of migration must be kept as low as possible. Manually applying the dynamic load balancing strategy to a given code that has already been parallelised can be time consuming, which is why it is desirable to automate the process. This requires that the strategy is generic if it is to be automated within a parallelisation tool such as CAPTools.

- *Develop a generic, minimally intrusive, effective DLB strategy for structured mesh codes.*
- *Develop utilities (library calls) to simplify the implementation of such a strategy.*
- *Transform existing message passing SPMD code.*
- *Automate this DLB strategy within the CAPTools environment.*

Figure 1.8: The four key aims of this research.

The fact that this research aims to devise a generic DLB strategy means that the approach used need not be exclusively related to CAPTools. Even if an application code has been parallelised using an alternative method, such as using the KeLP framework [22] for example, the DLB strategy discussed here should still be applicable if a similar parallelisation strategy has been used (see Appendix A).

1.16 Summary

This Chapter has discussed the benefits and drawbacks of parallel processing, highlighting the significant effect of load imbalance on parallel efficiency. Examples were used to demonstrate the different classifications of load imbalance, defining processor and physical imbalance. Various solutions to the problem of load imbalance were considered from which dynamic load balancing was found to be a suitable proposal, hence the assessment of a number of existing dynamic load balancing strategies. The possibility of automation was also deliberated, leading to an evaluation of manual parallelisation, parallel compilers, and parallelisation tools, where it was decided that a dynamic load balancing strategy would be automated using a tool.

Appendix A aims to give an insight into the parallelisation technique and the communication libraries that are used within CAPTools (a parallelisation tool), providing a basis for the in-depth coverage of CAPTools in Appendix B. Using the background knowledge of CAPTools from these two Appendices, several possible dynamic load balancing strategies are described in Chapter 2, where the selected strategy is analysed in more detail and the actual load balancing technique is explained. The utilities needed to implement the dynamic

load balancing strategy are given in Chapter 3, which are then used (tested) in the manual implementation of the strategy in Chapter 4. The automatic implementation of the dynamic load balancing strategy is then detailed in Chapter 5 using the CAPTools algorithms and data structures discussed in Appendix B, after which Chapter 6 covers the matter of load balancing unstructured mesh codes. Conclusions are given Chapter 7, along with raised concerns requiring further work, and future issues and discussions.

Chapter 2 The Dynamic Load Balancing Strategy For Structure Mesh Codes

The need for DLB was illustrated earlier in Chapter 1, where it was shown that load imbalance can have a detrimental effect on the parallel performance of an application code. The initial distribution, in which each processor is allocated an approximately equal workload, is not suitable for all types of problems, and so the distribution needs to be changed in order to improve the parallel performance. The maximum processor time must be reduced to improve the efficiency of an imbalanced code, which can be achieved by redistributing the load when the level of imbalance becomes significant, enabling the processors to finish computing in the same amount of time and consequently reducing the idle time. The basic idea behind redistribution is to migrate the workload off heavily loaded processors onto other neighbouring processors with a lighter load. This Chapter will compare various DLB strategies for structured mesh codes, after which the selected DLB strategy will be examined in detail, using CAPTools terminology where necessary, since the selected strategy is to be automated within this parallelisation tool. The user has no control over the factors involved with load imbalance, implying that DLB can be implemented in any structured mesh code (parallelised by CAPTools) if the overhead associated with its operation is negligible (in the case where imbalance exists).

2.1 Goals For The Dynamic Load Balancing Strategy

A number of goals need to be satisfied when developing a DLB strategy (Figure 2.1). The DLB strategy should be feasible to manually implement and understand, otherwise it may not be possible to automate its implementation within CAPTools. As with any working code, it is essential that the user is able to understand their code in order to maintain and optimise it, therefore minimal changes should be made to the user's code. Any code that is inserted should not

be obtrusive, and should be distinguishable from the original parallel code, implying the need to use utilities whose underlying operations need not be known by the user. The amount of inserted DLB code should be small in comparison with the original parallel code.

The DLB strategy needs to be generic if it is to be applicable to a wide range of codes (see criterion of CAPTools in Figure 1.3), therefore the strategy should not be developed for a specific application. The DLB strategy should be applicable to any structured mesh application code that has been parallelised by CAPTools, otherwise its functionality within CAPTools would be restricted. The DLB strategy should comply with those goals specified for CAPTools (Section 1.8), such that the strategy should be efficient, scalable, and portable.

The DLB strategy should obviously improve parallel efficiency (by reducing the maximum processor timings, and consequently the idle time), otherwise it would be pointless to implement such a strategy. The quality of the balance attained should be reasonably good (as achieving perfect balance may not be possible), where the effects of processor and physical imbalance are taken into consideration. The DLB strategy should be flexible enough to handle more than one specific type of load imbalance, ensuring that the load is redistributed regardless of the cause of imbalance.

The load should be redistributed when the cost of redistributing the load is less than the cost of the load imbalance [48, 86]. The load should be changed when possible and when profitable, implying this decision should be made every iteration since the load imbalance may continuously change during execution. A simple algorithm should therefore be used to calculate the new workload, which is cheap to perform in order to avoid dominating the overall execution time. The new distribution should only be implemented if enough cells are to be moved, as the load may oscillate due to the granularity of the structured mesh problem where single cells cannot be moved (see Section 2.2), implying that the 'optimal' load may not be attainable. Some form of damping should preferably be used to avoid load oscillation, where it is better to underestimate the new load rather than overestimate it to avoid the unnecessary movement of data. The data can be moved in a subsequent redistribution, meaning that if the new load is underestimated then the remaining load will be moved in the next distribution,

whereas the load may have to be moved back to the owning processor if the new load is overestimated.

Communications are used to migrate the data, meaning data movement should be kept to a minimum to avoid the communication costs rising. The DLB code should show an improvement over the non-DLB code, which is a reason why the time to migrate the load should not dominate the overall execution time. The data should be moved gradually, rather than in large amounts, where data is ideally only transferred between neighbouring processors to avoid major changes to the user's code, and to reduce communication overheads (see Section 1.14.3).

1) The DLB strategy should be feasible (the user must be able to implement it manually)
2) Minimal changes should be made to the user's code
3) The strategy should be applicable to a wide range of codes
4) The DLB code should be efficient, scalable, and portable
5) Improve parallel performance
6) The quality of the balance should be reasonably good
7) The strategy should consider the effects of processor and physical imbalance
8) The load should be redistributed according to processor capability
9) Distribution should be changed when possible and when profitable
10) The algorithm to calculate the new workload should be cheap to perform
11) Load oscillation should be avoided if possible
12) The number of additional communications should be kept to a minimum
13) Data movement should be kept to a minimum

Figure 2.1: Goals for the DLB strategy.

2.2 The Importance Of Retaining A Rectangular Partition

The parallel efficiency of a code may be poor due to load imbalance, which can be improved upon with the use of load balancing. The load is 'balanced' by redistributing the workload, which essentially means having to change the processor partition range limits (Section A.2) and update the processor ownership of any distributed data though communications.

Rectangular partitions are used within CAPTools (Section A.2.1) and so it would be beneficial to take advantage of this fact when devising the possible DLB strategy, as the new load is determined by changing the partitions. As stated in Section 2.1, one of the main requirements of the DLB strategy is to minimise changes to the user's code, and so using a non-rectangular partition would not meet this requirement. If a non-rectangular partition were to be used then loop limit alteration would no longer be sufficient to implement the 'owner computes' rule (see Figure B.45 in Section B.8) within the code. The original loop would need to be duplicated and then processed over the different sections of the non-rectangular workload. It may be necessary for a processor to communicate with several neighbours in any given direction, and sometimes with the same processor when communicating in different directions, which does not follow with minimising the communication latency. If the partition were changed such that a non-rectangular partition is used, then this would involve changing the source in the parallel code. When balancing the load it is therefore necessary to retain a rectangular partition in order to improve parallel performance, without incurring high communication latencies or major alterations to the source code. The main benefit of retaining a rectangular partition is that most of the parallel code remains the same, only a small proportion of the code actually needs to be altered.

An optimal load balance may never be attainable with DLB for structured mesh codes as an entire row, column or plane of cells may be moved, which is unlike the movement of single cells with DLB for unstructured mesh codes (see Chapter 6).

2.3 Static Load Balancing Strategies

Most parallel codes are balanced statically such that each processor has an equal workload, where the initial processor partition range limits are not changed during execution. Once the processor workload has been specified there is no way of altering the load, which is unsuitable for situations in which the load imbalance is changing continuously throughout execution for example. There is no way for the user to make an accurate prediction on how the load should be balanced, as the

user has little control over any external factors, such as the processor speed or computational workload.

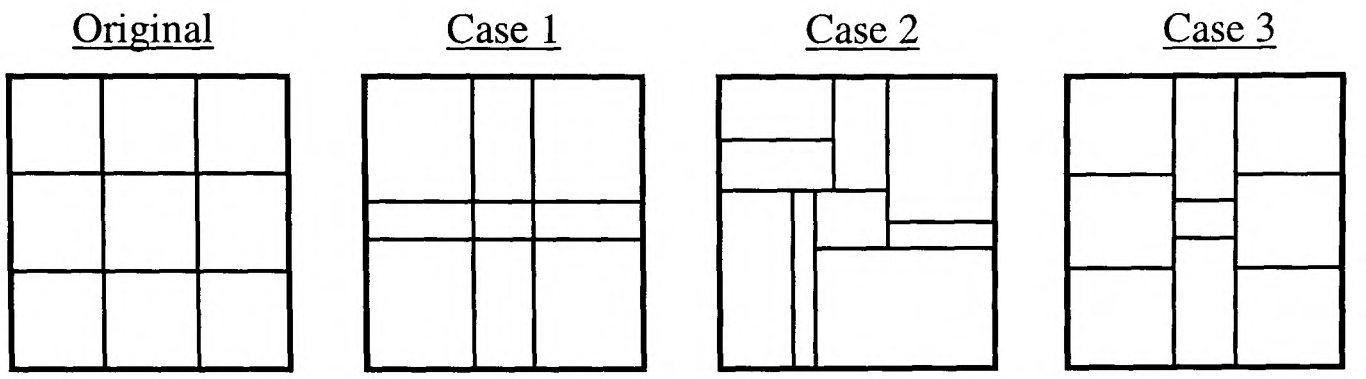
CAPTools is a generic tool, partitioning data evenly across the specified number of processors because the partition cannot be specific to one type of problem. CAPTools has no knowledge relating to processor speeds, or number of users, or how the code will behave during execution, and so it cannot determine how the load should be distributed, which is why it assumes that the workload should be the same on every processor. However, it has been shown that the initial partition is not always suitable, and so the workload should be changed.

Although it has already been decided that the load will be balanced using a dynamic approach, the following explains in brief how to implement a static load balancing approach. Note that at present, the processor partition range limits can only be changed globally (remain coincidental) since current CAPTools communication utilities are only capable of handling this type of situation. The processor partition range limits can be changed manually after the call to CAP_SETUPDPART, which sets up the initial distribution, where the user will be able to specify how to distribute the workload, after which each processor will operate on their defined range. This method of balancing the load is cheap, since no calculations are needed to determine how to distribute the load at runtime, and it is not necessary to migrate the load (since each processor already owns the load that they operate on from the onset). No major changes to the user's code are required with this method, however this approach will not be successful at balancing a wide range of application codes.

2.4 Dynamic Load Balancing Strategies

Three different load balancing strategies, shown in Figure 2.2, shall be examined in the context of structured mesh code problems, each trying to achieve a good load balance without incurring high communication costs or major alterations to the source code. The majority of communications should only occur with immediately neighbouring processors to help maintain low communication costs and reduce the changes to the user's code.

The processor partition range limits that define the workload shall be changed during execution based on some level of load imbalance, where the load will then have to be migrated onto the new owner of the data. The DLB strategy should be relatively easy to implement, attaining a reasonably good load balance without incurring too many overheads, where the user should still be able to recognise, maintain, and optimise their DLB parallel code.



Case:	1	2	3
Limits:	Change globally	Change locally	Mix
Communications:	No change	Complex	Relatively simple
Balance:	Moderate	Very Good	Good

Figure 2.2: Three different load balancing strategies are shown, comparing each against the original distribution in which the load is distributed evenly.

2.4.1 The Initial Problem

The Original problem, shown in Figure 2.2, shows a 2D mesh that has initially been evenly distributed onto 9 processors using a 3x3 grid topology, where global (coincidental) processor partition range limits are used in every dimension. Assuming that the middle processor hinders the parallel efficiency of this problem (i.e. it is the slowest), then its load needs to be redistributed onto neighbouring processors. It is expected that the load on the middle processor will be reduced and placed onto neighbouring processors.

2.4.2 Case 1 – Coincidental Processor Partition Range Limits

The first strategy, represented graphically in Figure 2.2, tries to improve upon the current load imbalance by changing the partition range limits globally in each dimension (i.e. each dimension contains coincidental limits). The Left/Right and Up/Down limits are squeezed inwards to reduce the load on the middle processor, retaining the communication structure that ensures that only immediate neighbour communications are necessary. Because the global limits are still guaranteed, load migration and the coding of this strategy is relatively simple in comparison to the cases that follow, but the balance attainable is only moderate because of the use of global limits. The problem with this strategy is that the load is also reduced on the four immediately neighbouring processors surrounding the middle processor, irrespective of whether they needed to be or not. It is plain to see that the load balance attainable using this strategy is limited by the inflexibility in having to use global limits in which the partition range limits are forced to coincide with those on neighbouring processors.

2.4.3 Case 2 – Non-Coincidental Processor Partition Range Limits

The second strategy represented graphically as Case 2 in Figure 2.2, uses non-coincidental partition range limits (local limits) in every dimension, allowing flexibility in attaining a very good load balance since the new load is less constrained by the load on a neighbouring processor. Although a rectangular partition is still utilised here, the main concern is that it can be very difficult to ensure that there are no "gaps" when constructing the partition and so bisection may be needed in order to calculate the new partition that must still map onto the processor topology. Another problem with Case 2 is that due to the usage of local limits a processor may have a number of neighbouring processors in any given dimension. For example, in Figure 2.2 the middle processor now has 6 neighbouring processors instead of just 4 neighbouring processors. This means that when communicating in a particular direction a processor may no longer be

communicating with an 'immediate' neighbour but with several neighbours, increasing the communication overhead. This complex communication structure makes it very difficult to code (although not impossible), particularly in relation to how processors communicate with one another, and major changes to the user's code may be needed to implement this strategy.

2.4.4 Case 3 – A Combination Of Case 1 And Case 2 ('Staggered Limits')

In the third strategy, shown in Figure 2.2 as Case 3, one partitioned dimension uses local processor partition range limits and the remaining dimensions use global limits, giving the impression of 'staggered limits'. The balance attainable using this approach is better than in Case 1 because the local limits have made the balance more flexible. In addition the communication latency is not as high as in Case 2 due to the fact that some global limits have been used. Communications in the dimension containing the non-coincidental limits (those that appear 'staggered') remain with immediately neighbouring processors whereas orthogonal communications will need to change. The expectation is that the coding of this strategy is fairly simple, and it remains recognisable to the user (see Section 4.8). Note that Burgess [85] and Cermele et al. [68, 69, 77 and 82] make use of this type of partition.

2.5 The Selected Dynamic Load Balancing Strategy

Case 1 forces all partition range limits to coincide with those on neighbouring processors, greatly restricting the load balance possible as the workload decrease required on one processor is restricted by the workload increase or decrease required on a neighbouring processor. Although Case 2 allows for good load balance when using all non-coincidental partition range limits, it suffers from complicated communications and difficulties in constructing the partition.

However, Case 3, where partition limits are forced to coincide on all but one dimension, allows for good load balance as well as fairly simple and neat communication patterns, and is relatively straightforward to construct, and is therefore selected for the generic strategy.

2.5.1 The DLB Staggered Limit Strategy

In Case 3 in Figure 2.2, the Up/Down limits appear to be 'staggered', and so for this reason the dimension containing non-coincidental limits shall be referred to as the Staggered Dimension and the remaining dimension(s) shall be known as the Non-Staggered Dimension(s). Also note that it is now possible for a processor to have several neighbouring processors in the Non-Staggered Dimension(s), and so new issues, such as non-neighbour inter-processor communication and load migration, need to be addressed before trying to implement the DLB strategy.

2.5.2 The DLB Communication Structure

Originally each processor only had to communicate with an immediate neighbour in any partition dimension (Section A.3.2), but now each processor may have to communicate with several 'neighbouring' processors in an adjacent 'block of processors' when communicating in a Non-Staggered Dimension. For example, in Figure 2.3 representing the new communication structure of Processor 14 for a 3x3x3 processor topology in which the second dimension is said to contain the staggered limits (compare with Figure A.12 showing the original communication structure), each block contains 3 processors. Every processor in a block shares the same Non-Staggered Dimension processor partition range limits as the other processors in that block. For instance, Processors 11, 14, and 17, all share the same Left/Right and Back/Forth limits (which are not staggered). A processor can communicate with any processor in an adjacent block, but need only communicate with an immediate neighbour when communicating in the Staggered Dimension.

In Figure 2.3 for instance, Processor 14 (and likewise for Processors 11 and 17) can potentially communicate with Processors 10, 15 and 16, when communicating to the Left since they are contained within the adjacent block in that direction. Similarly, Processor 14 can potentially communicate with Processors 2, 5 and 8, when communicating in the Back direction. However, Processor 14 still only needs to communicate with Processors 11 and 17 in the Up/Down direction.

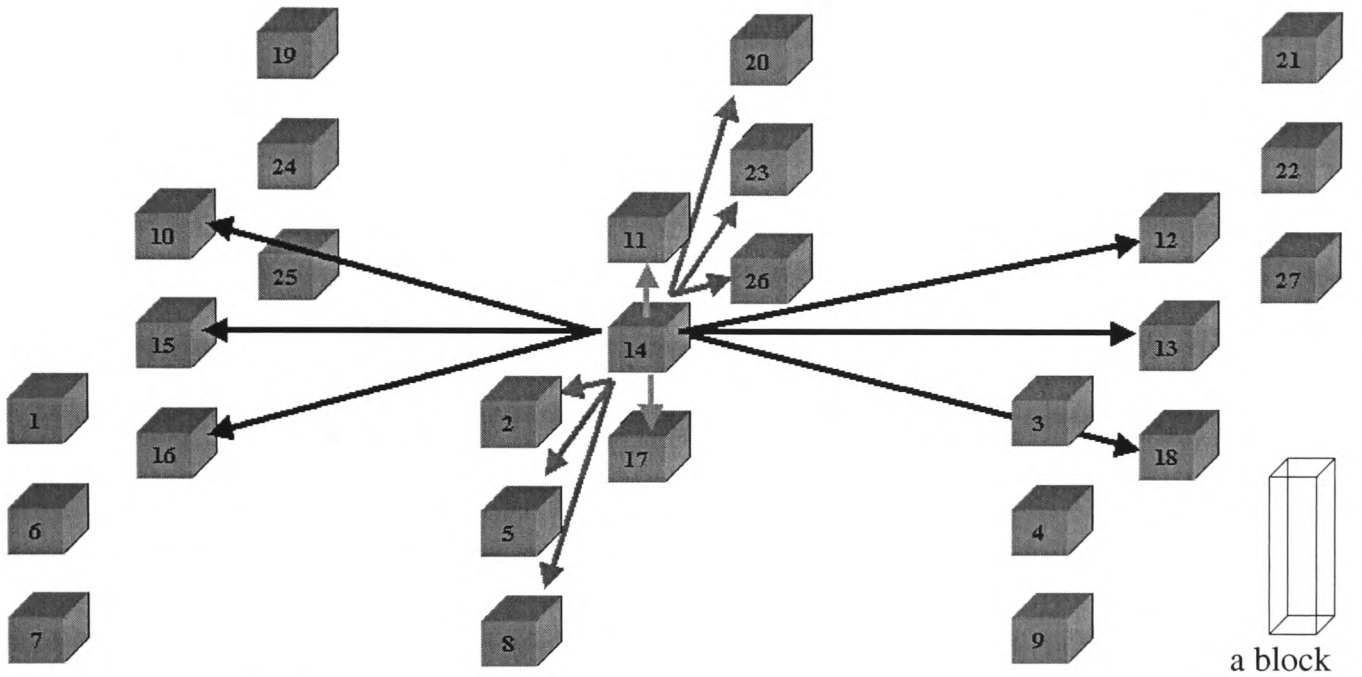


Figure 2.3: Shows the processor communication structure used for the selected DLB strategy in a 3D-grid topology, where the Staggered Dimension processor partition range limits are in the Up/Down direction. The neighbouring processors of Processor 14 are indicated for each direction where a 'block' contains a group of neighbouring processors that share the same limits in the Non-Staggered Dimensions.

2.5.3 Inter-Processor Communication

Figure 2.4 shows a 3D mesh mapped onto 27 processors, with Staggered Dimension processor partition range limits used in the Up/Down dimension. Communications in the Staggered Dimension remain the same with immediate neighbour communication, however, the staggered limits affect communications in the Non-Staggered Dimensions (Left/Right, and Back/Forth in this case) meaning that data may be needed from several processors in a Non-Staggered Dimension. For example, in Figure 2.4 Processor 6 will still only need to communicate with Processors 1 and 7 (immediate neighbours) when updating the

halo regions in the Staggered Dimension. Whereas Processor 6 will need to receive data from Processors 2, 5, and 8, and not just from its immediate neighbour (Processor 5), when updating its Right halo region. Note that it is not always necessary to have to communicate with all of the potential neighbours, as can be seen in the situation where Processor 2 is sending data to its Right, whereby only Processors 3 and 4 need to receive data, and no communication occurs with Processor 9.

Therefore the existing types of communication calls (Section A.3.3) are not solely applicable when the chosen DLB strategy is used, as a processor can communicate with potentially several neighbours now and not just its immediate neighbour. This can be demonstrated more clearly by comparing the update of halo regions using the original communication structure in Figure A.8, and using the new communication structure (in 2D) in Figure 2.5.

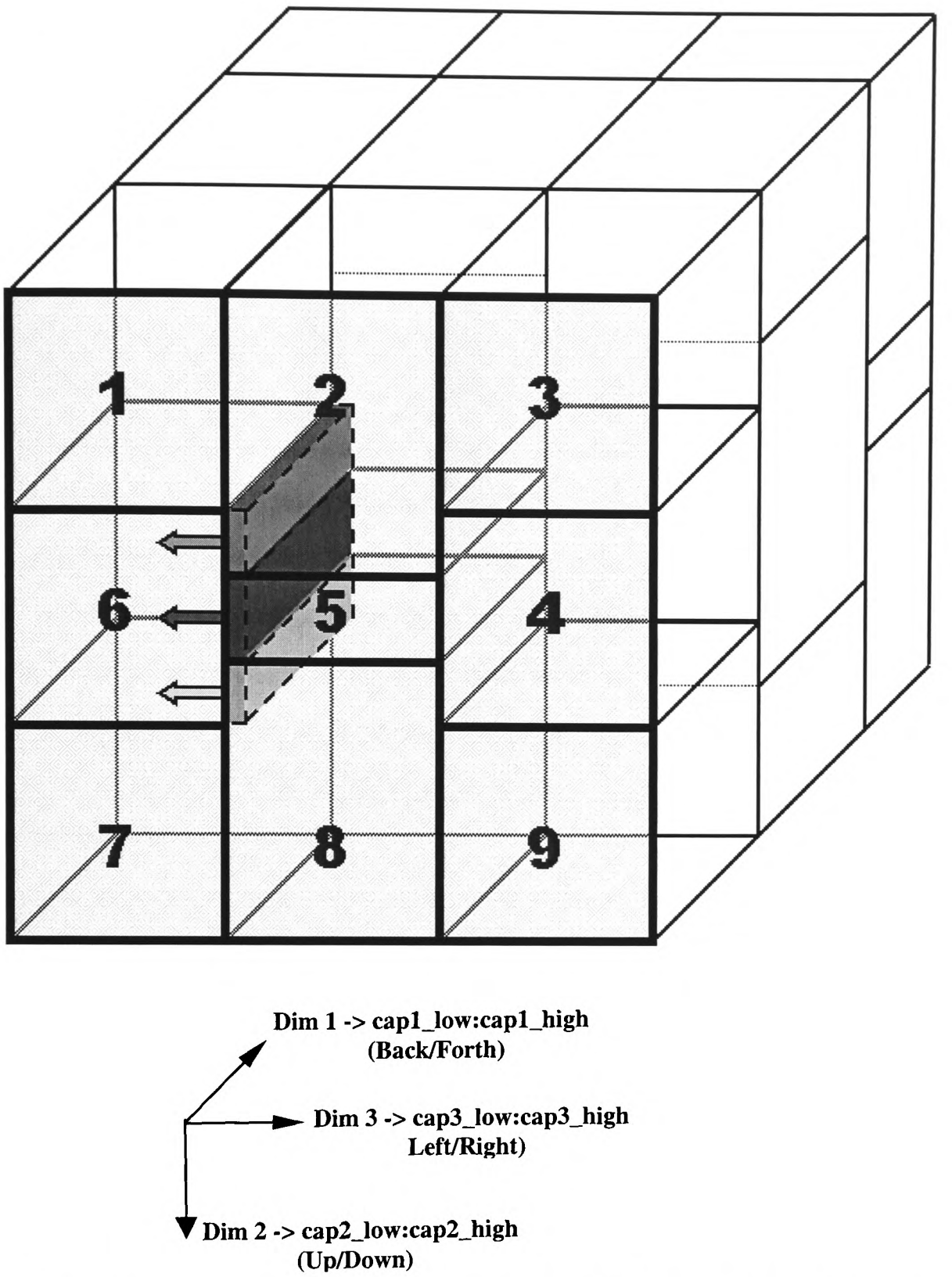


Figure 2.4: Shows a mesh of processors containing local limits in the Up/Down direction, highlighting the instance in which Processor 6 receives data from its Right. Dimension 2 is the Staggered Dimension, implying that dimensions 1 and 3 (the Non-Staggered Dimensions) use global processor partition range limits.

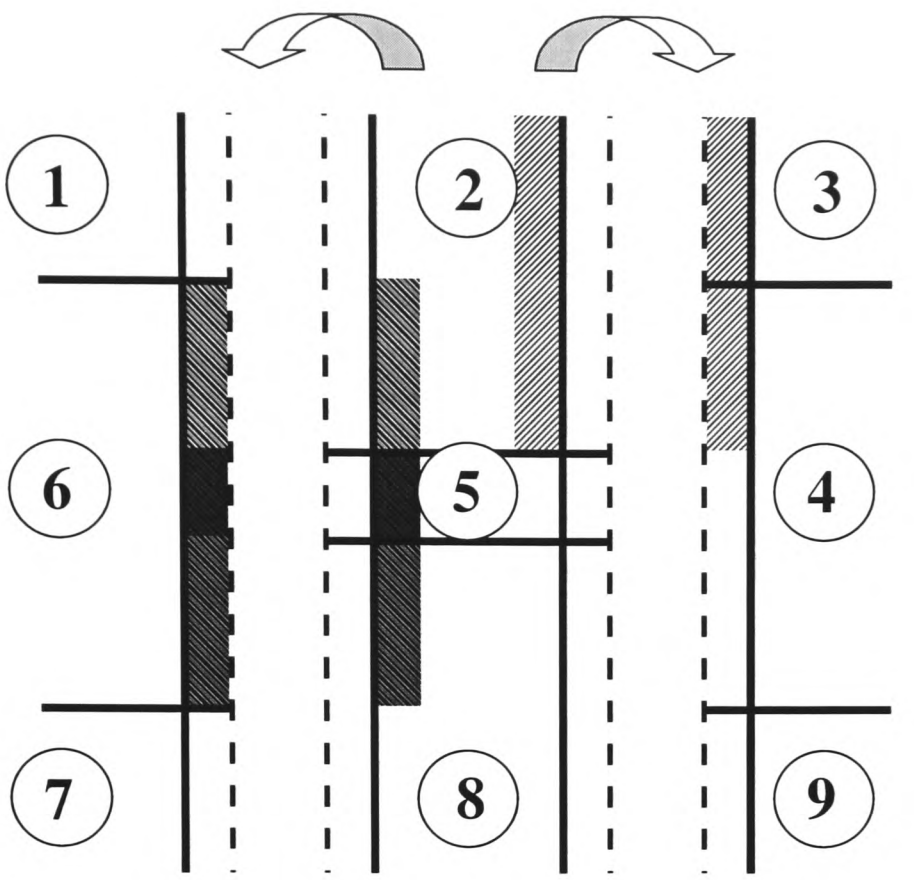


Figure 2.5: A 2D representation of the halo update on Processor 6 from several neighbours, and not just their immediate neighbour. Also shown is the data that is sent from Processor 2 to its neighbours on the Right.

When communicating in a Non-Staggered Dimension the original communication message may need to be dissected in the Staggered Dimension, and so it is first necessary to determine the set of neighbouring processors involved in the communication. Once a neighbouring processor has been detected the amount to communicate can be calculated by obtaining the overlap between the staggered limits of the processors involved. If the staggered limits do not overlap then the amount to communicate will be zero, in which case a communication with this neighbouring processor is disregarded and the same process is applied to the next neighbouring processor in the given direction.

Most communications will occur between the processors' staggered limits, but on occasions the communication may extend beyond these processor partition limits, as seen in Figure 2.6. In such instances all processors will be communicating beyond their own limits, and so when comparing the staggered limits these 'offsets' need to be considered. For example, the starting address of the communicated variable may no longer be the lower processor partition range limit (Section B.9.1). If the offsets are not considered then it is possible that the correct operation will not be undertaken, since some processors shall not receive

some of the data that they requested. The same operation must occur in the DLB parallel code as in the non-DLB parallel code.

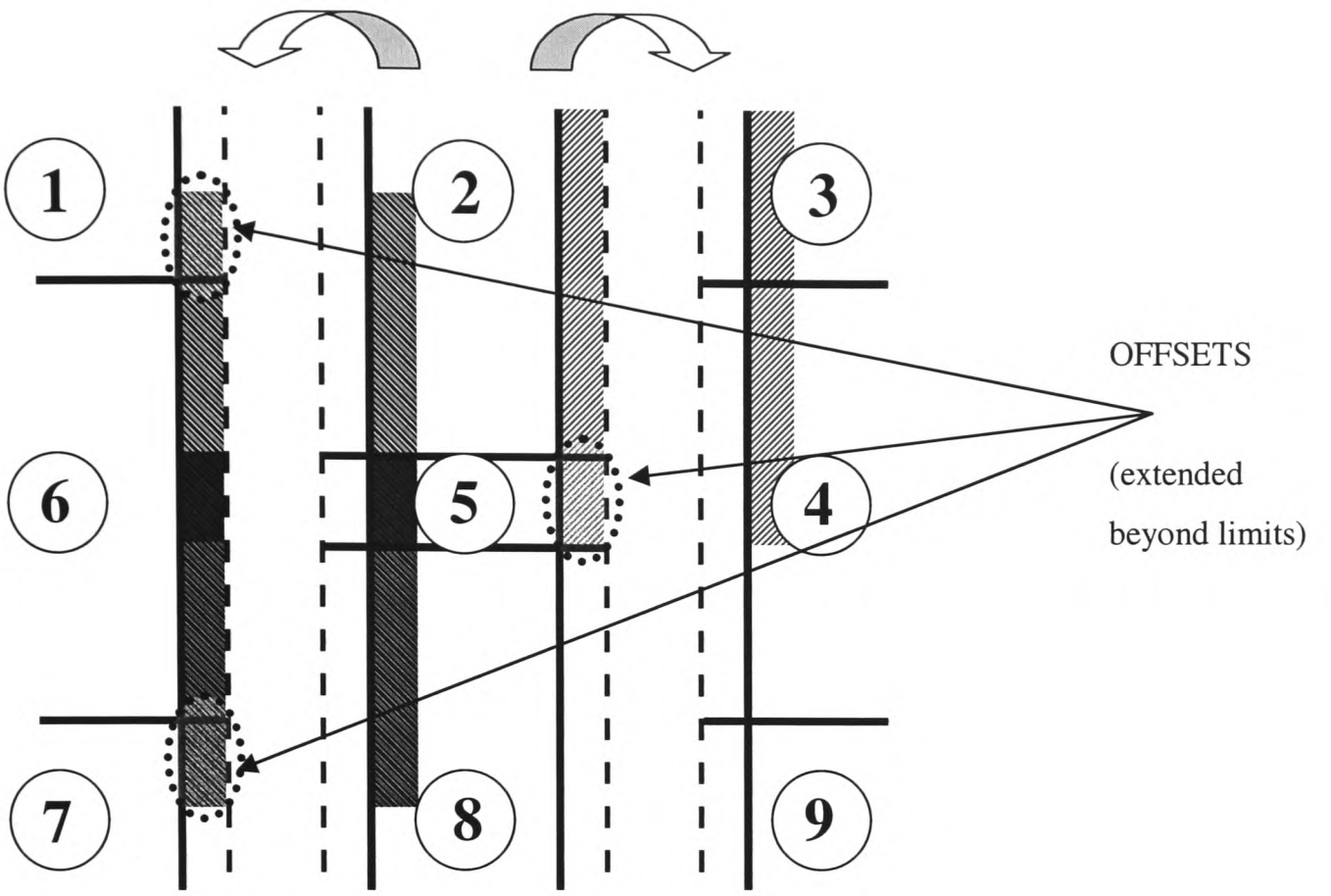


Figure 2.6: A 2D example illustrating what happens when the communication extends beyond the processor partition range limits, on Processor 2 and Processor 6. The 'offset' data (from the processor limit) must be included in the communication.

When a communication is executed within an execution control mask then this means that only those processors who own the data will be involved in the communication. Unlike above, the communicating processor will not need to communicate with several processors but just a single processor whose staggered limits contain the control value. For example, in Figure 2.7, those processors that own the N^{th} row of data will need to communicate.

```
IF (N.GE.CAP2_LOW .AND. N.LE.CAP2_HIGH) THEN
   CALL SEND(A(1,N),10,2,CAP_LEFT)
END IF
```

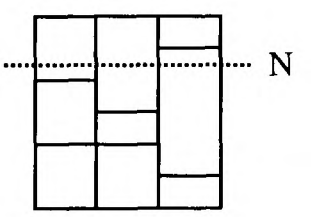


Figure 2.7: Example illustrating a communication that is executed within an execution control mask.

To minimise changes to the user's code it is desirable to use a single call (DLB communication) to internally allow processors to communicate with several

possible processors, and not have several individual communication calls in the application code. The call should appear similar to existing communications, so that the user should still be able to understand the underlying communication, whilst also recognising its purpose, i.e. it should be distinguishable from normal communications. The call should therefore still be specified in a particular direction, as before, and not to an explicit processor, allowing the utility routine to internally determine which processor(s) to communicate with. It would be difficult to hard-code which neighbour a processor should communicate with as this can change after each redistribution (in which the staggered limits may be changed), which is another reason why a generic utility should be used, automatically determining whom to communicate with at runtime (see Section 3.2.1).

2.6 Load Migration

Load migration is a fundamental component of this DLB strategy, as it is where data is transferred from one processor to another via a set of communication calls, to construct a new partition with the aim of improving the load balance. Looking back at the load balancing strategies described earlier, a drawback of using all local partition range limits (Case 2) is that load migration would be extremely complex (involving many communications). However, the load migration of the selected strategy may appear complex, but in fact it is relatively simple. The reason for this is that even though several neighbours may be involved when migrating in a Non-Staggered Dimension, communications will only involve immediate neighbours when migrating in the Staggered Dimension. Once again, to attempt to minimise the changes to the users' code, generic utilities can be used to migrate data in a given direction, one to migrate data in the Staggered Dimension and the other to migrate in a Non-Staggered Dimension. The issue of load migration shall be dealt with in more detail in the next Chapter.

2.7 DLB Issues

A number of issues need to be addressed in order to implement the selected DLB Staggered Limit Strategy (Section 1.14). Identifying the load imbalance and deciding when and where to change the partition and redistribute the workload must be determined during runtime. Any utilities created to do these should be generic, particularly since they are to be automated within CAPTools, and since one of the key goals is to attempt to minimise the changes to the users' code. The fact that this strategy contains staggered limits should not be neglected but integrated into the following specifics of dynamic load balancing.

Even if there is no load imbalance, as long as the overhead associated with the DLB communications is very small (Section 3.3.5) then there is no reason not to run in DLB mode. One advantage of choosing to execute the parallel code in DLB mode is that the user may not always be certain that there will be no load imbalance. Additionally, implementing the DLB parallel code rather than the non-DLB parallel code allows the user to execute their problem on any heterogeneous system of processors, whereas the user is restricted with respect to the processor specifications when running the non-DLB version.

2.7.1 Where To Redistribute The Workload

The user may chose to run their parallel code in DLB mode having some initial suspicion that load imbalance exists within their code. The user may want to dynamically load balance their code either having knowledge of the actual physical characteristics of the code, or knowledge of the processor characteristics in terms of processing speed or number of jobs. A profiler, or the user's knowledge of the code, can therefore be used to identify the exact location of any significant load imbalance that exists within the code, which can be used to determine where to redistribute the load.

The load will most likely need to be redistributed within some sort of loop, such as a time-step, iteration or solver loop [87]. One example is shown in Figure 2.8. The ideal location to DLB the code is in a loop that is iterated many times by

each processor, as the load imbalance is magnified in proportion to the number of iterations. The load could be imbalanced in numerous locations within the code, and so it would be beneficial to the user to know this information so that the dynamic load balancing code can be placed at the different levels of granularity. Having determined the location of code containing the load imbalance the next decision to be made is where exactly to redistribute the load within this location. Should the load be redistributed at the beginning, during or at the end of an iteration, and is this issue of any significance? In terms of manually implementing the DLB strategy with staggered limits it does not make a difference whether the load is redistributed at the end of an iteration, or at the beginning of the next iteration, since the operation performed is the same. However, in terms of automating the placement of the DLB code the issue of placement has some bearing on the ease of automation and is explained further in Section 5.7.

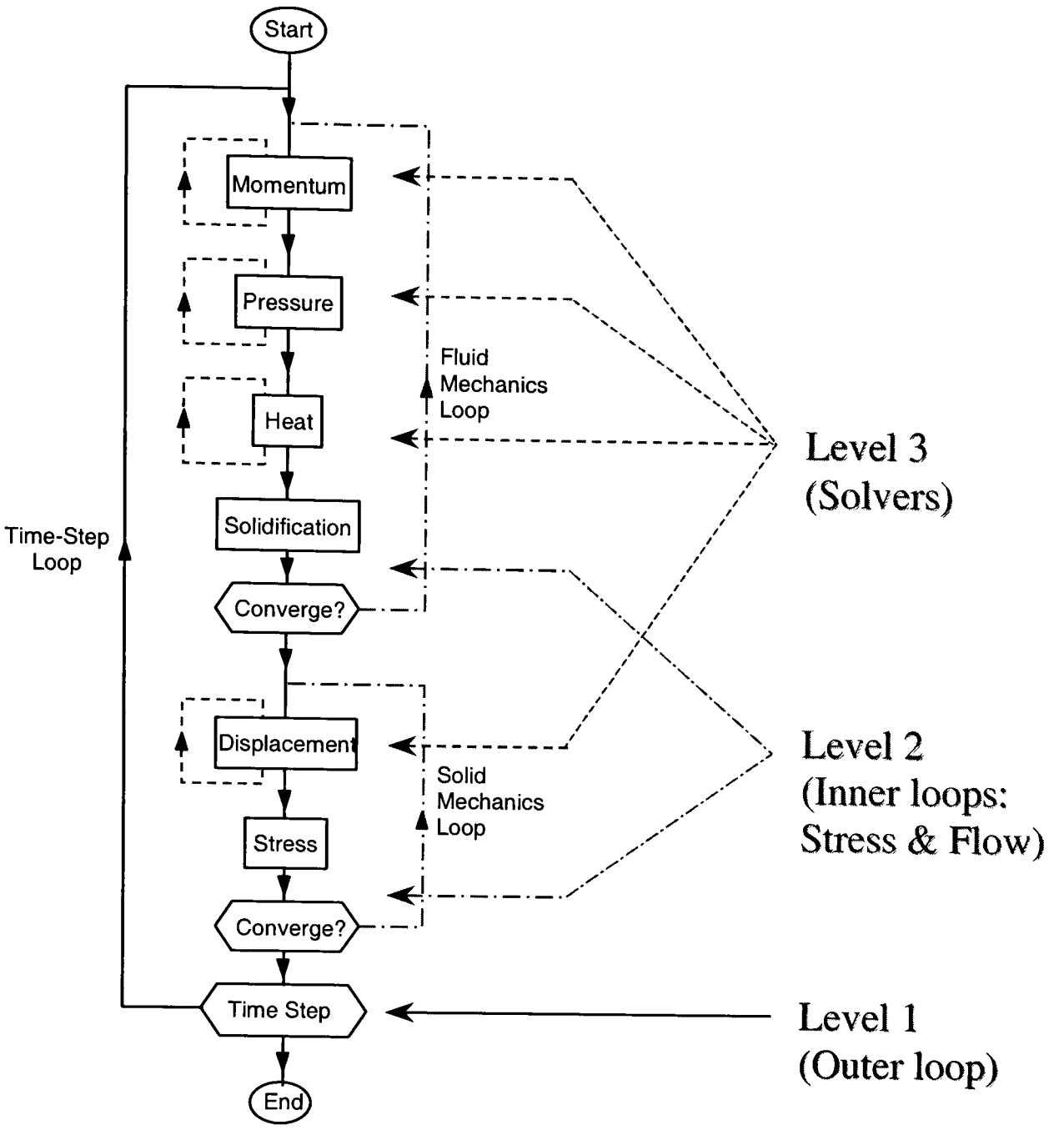


Figure 2.8: Example illustrating the different levels in which the load could be balanced.

2.7.2 Frequency Of When To Redistribute The Workload

Section 1.14.2 elaborated on previous research into the issue of determining when a load redistribution is needed. Effective determination of when to redistribute is crucial if the DLB is to be profitable. If the load were not balanced then the idle time would dominate the overall performance of the parallel code. Balancing the load occasionally could lead to some improvement, but the idle time would still be quite significant, whereas at the opposite end, balancing too frequently could lead to the redistribution time becoming dominant. It is important to balance the load without hampering the performance of the code being balanced by taking into

account certain factors. If static load balancing were used then there would be no need to decide when to balance the load, as the load would not be redistributed during execution, but would be balanced just once. The load could be balanced a set number of times, where either the user specifies how many times, or a default value is used. The problem with this idea is in deciding how frequently to balance the load, for example, at what iteration should the load be balanced? This value is unique to the problem being solved, and a default value, or user specified value, cannot be applied to all runs of the codes correctly, whereby the load is redistributed at the optimal iteration. For example, if the user specifies that the load should be balanced 100 times, 10 iterations apart, then this would not be suitable if the problem could be 'balanced' in the first 5 iterations without the need for any further redistributions. The user would have overestimated the number the redistributions that were necessary, as well as the interval between them, allowing the redistribution time to become significant. If the opposite had been true, where many redistributions were required over quite a large time span, and the user had underestimated these figures, then the load imbalance would continue to hinder the performance of the code (where idle time is significant).

It has been decided that the frequency of when to balance the load shall be determined at runtime, due to the difficulty in predicting how many redistributions will be sufficient, and the intervals between them. During each iteration it can be decided whether or not the load should be balanced at this iteration, based on some 'measure of load imbalance'. This means that the load can be balanced if required and only when it is proved profitable to do so.

2.7.2.1 The Influence Of Processor And Physical Imbalance

It is important to be able to distinguish between the different instances of load imbalance since these factors affect the way in which the load is redistributed. The load is transferred from the slow/heavily-loaded processors onto the fast/lightly-loaded processors. With processor imbalance the load is reduced on the slow processors and placed onto faster processors, who process these additional cells at their own rate. With physical imbalance the load is reduced on the heavily-loaded

processors and placed onto the lighter-loaded processors, who process these additional cells at the rate of the gained cells. A more complex situation can arise with a combination of both instances, for example, when a heterogeneous system of processors is used to solve a physically imbalanced problem in which some physical phenomena is occurring on a geometrically imbalanced structure.

The way in which the new limits are calculated depends on the type of problem, as cells are either gained at a processors own rate of processing a cell, or at the rate of the losing processor. This factor cannot be ignored, otherwise the new load would be redistributed incorrectly.

With processor imbalance the load should be fairly well balanced after redistribution, as the variation between processors has been catered for in the new distribution. If more than one job can be run on a processor, at any instance in time during the execution of the users' code, then a number of redistributions may be necessary, but these should typically occur when jobs are added to, and removed from, a processor.

With physical imbalance the load is balanced according the current instance of load imbalance, which can change continuously throughout the execution. This suggests that it may be impossible to have a set number of redistributions that will guarantee that the load will be fairly balanced, as the load keeps changing due to the physical characteristics of the code. It is assumed that the load changes over time, but that the load does not change dramatically from iteration to iteration. Therefore, the load imbalance in one iteration will be approximately the same in the next iteration, implying that if the load is balanced in one iteration then the resulting positive effect can be seen in the subsequent iteration(s). For example, if some physical phenomena occurs on a single processor at iteration 5, where the load is then redistributed, then there should be less idle time present in iteration 6, where the load has been reduced on the heavy processor. As the load changes again then further redistributions are needed, which depends solely on the application code, justifying the reasons behind not fixing the number of redistributions before runtime.

Note that if both processor and physical imbalance exist together within the code then it may be difficult to determine how much of each factor is attributing to the current instance of load imbalance. This may confuse the issue of when to balance the load, and how to adjust the timings (Section 3.5.3).

2.7.2.2 A Model To Predict When To Redistribute

Each time the load is redistributed, data is transferred from one processor to another, which has a cost associated with the whole process. It is assumed that in subsequent redistributions only slightly less data shall be migrated, and so it is safe to overestimate the time of the next redistribution as being equivalent to the current redistribution time. If the redistribution time is significant then the next redistribution should be delayed until it is profitable to do so. If the next redistribution is not delayed then the redistribution time can dominate the overall execution time, which is undesirable. However, if the idle time is allowed to increase, due to continuing load imbalance, then this too impedes the parallel performance of the code, which is unwanted. Therefore, if the idle time becomes significant and the redistribution time is not too large, then it will prove profitable to redistribute the load at the current iteration. Hence, if the redistribution time is not significant then it would cost very little to perform a load redistribution in order to improve the efficiency of the code.

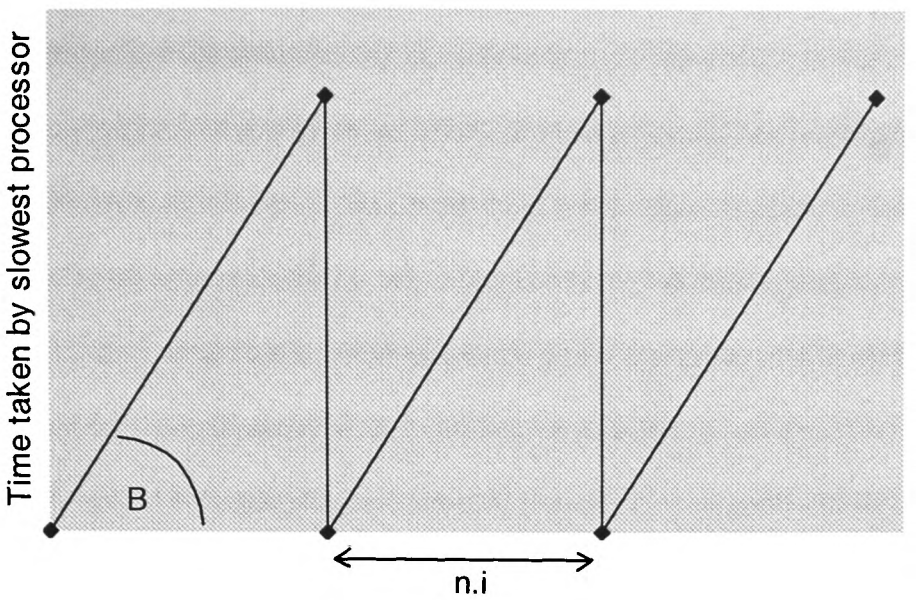
A model of computation can be used to determine how frequently to redistribute the load [87], as seen in Figure 2.9, which has several simplifying assumptions. Firstly, it assumes that the rate of increase in imbalance is linear in time, and secondly, it assumes that the entire load imbalance is removed by redistributing the workload (which is rare). An estimate for the time required to redistribute the load (i.e. calculation of new distribution and data migration) is also used based on previous history (i.e. previous load redistribution time), where initially the redistribution time is taken as, perhaps, a small percentage of the time for the first iteration, such that the first redistribution will occur soon. The model of computation is used to determine when the next redistribution should occur (in relation to when the previous redistribution occurred in terms of iterations of the selected loop), which can be calculated using the rate of increase of imbalance and the estimated time required to redistribute the load. The determination of when to redistribute the load needs to be simple and cheap, since there would be little use in spending a notable amount of time calculating when the next redistribution should occur, especially if the load is going to be redistributed several times, and also because this calculation will be performed every iteration of the imbalanced

loop. The cost of redistributing the load should be compared to the cost of not redistributing the load (which will allow the imbalance to grow), and so redistribution should be delayed until a future iteration if its cost is greater.

The graph in Figure 2.9 pictorially demonstrates the increase in load imbalance with redistributions removing all load imbalance (illustrated by the vertical lines). It is based on the assumptions previously mentioned and upon the measurements taken at the current instance. The time per iteration of the loop being balanced is i, and n is the number of iterations of this loop between rebalancing (i.e. what is being calculated). Therefore the time interval between rebalances is n.i. The gradient is the rate of increase of load imbalance (B), and the redistribution time is given as R.

From this graph, the idle time caused by load imbalance can be calculated by the average level of load imbalance ((n.i*B)/2), which can be integrated over time. Similarly, the cost of rebalancing (R) can be distributed over time by dividing by the interval between rebalances (R/n.i).

The aim is to calculate the number of iterations between redistribution (n) that minimises the overall time. This requires adding together the idle time cost and the redistribution cost, and differentiating with respect to n. This leads to the formula $n=\sqrt{(2R/Bi^2)}$ where t is minimised.



Key:
i =time per iteration
n =no. of iterations between redistribution
B=rate of increase of load imbalance
R=redistribution time
t =total time taken to redistribute the load

$$t_{cost} = \int_0^t \left(\frac{B \times n.i}{2} + \frac{R}{n.i} \right) dt = \frac{B \times n.i \times t}{2} + \frac{R \times t}{n.i}$$

$$\frac{dt}{dn} = \frac{B \times i \times t}{2} - \frac{R \times t}{n^2 \times i} = 0$$

$$\frac{d^2t}{dn^2} = \frac{2 \times R \times t}{n^3 \times i}$$ (implies a minimum)

$$\therefore n = \sqrt{\frac{2R}{Bi^2}}$$

Figure 2.9: Model of computation depicting load imbalance.

Examining the model of computation, it is evident that the value of n will increase as the redistribution time (R) increases, implying that the load will be redistributed later rather than sooner. Alternatively, the value of n will decrease if

either the rate of load imbalance (B) or the iteration time (i) increase, implying that the load will be redistributed sooner rather than later.

2.7.3 Measuring Load Imbalance

A suitable means of measuring the load imbalance for all of the situations, and for the model in Figure 2.9, is to actually time components of the parallel execution. The elapsed time of the imbalanced loop could be obtained, however, the timings on each processor would be the same because the elapsed time includes both the cpu time and the idle time, therefore this is not a suitable timer. Alternatively, the cpu time for the imbalanced loop could be obtained for each processor, however, on a multi-user system the idle time caused by other jobs running on the cpu would not be considered, again making this an inappropriate measure.

To overcome these problems the processor computation time can be obtained by finding the difference between the elapsed time of the imbalanced loop and the elapsed time of all the communications in the loop (which includes idle time). To achieve this each communication needs to be timed, this is done internally within the CAPLib communications [112].

Communication calls are in essence synchronisation points, which mean that when a processor reaches such a call they shall either execute the communication immediately or they may have to wait idle until the other processor(s) involved in the communication reaches the same stage. Those processors that are fast or lightly-loaded will reach the synchronisation point before their slower or heavily-loaded neighbours, and so they shall remain idle within this communication, indicated by a large time. Those processors that are slow or heavily-loaded will reach the synchronisation point after their faster or lightly-loaded neighbours, and so they will not need to wait at all, which is indicated by a small communication time.

2.7.4 Calculating The New Workload Distribution

The overall aim of DLB is to improve the parallel performance of the code, which can be achieved by reducing the overall execution time, which is, in effect, the maximum processor iteration time. Reducing the workload on heavily loaded processors can reduce the maximum processor time, which means shifting the load onto neighbouring processors, preferably onto those with a lighter load. The idle time is also decreased as a consequence, as the heavy processors have less work, and the light processors have more work, reducing the waiting time between all processors which utilises the available resources more effectively. Therefore the workload on some processors needs to be changed, which means changing the processor partition range limits that define the workload.

The load is only migrated onto a neighbouring processor because this ensures that the communication overhead is kept low. If the load could be shifted onto any processor then this would mean that each processor could potentially need to communicate with several other processors, forcing the communication structure to be changed. One benefit of only being able to shift the load onto a neighbouring processor is that the load can only be moved gradually and not all at once, which acts as a damping effect when calculating the new limits.

The new limits are calculated separately for each partitioned dimension, where the Non-Staggered Dimensions are processed before the Staggered Dimension. The local limits in the Staggered Dimension allow more flexibility when trying to achieve a better 'balance', which is why this dimension is balanced last so as to 'fine tune' the 'general' balance already obtained when balancing the Non-Staggered Dimension(s).

From Figure 2.10 it can be seen that the Left/Right limits need to be global (e.g. each processor in the middle column of processors have the same Left/Right limits), and that the Up/Down limits are going to be staggered. The individual processor timings are used to balance each individual column of processors in the Staggered Dimension, but we have to use the overall column times when balancing in the Non-Staggered Dimensions as this balance cannot be based on any individual processor, consequently producing a 'general' balance. Note that the new workload is governed by the granularity of the structured mesh code,

because unlike unstructured mesh codes, single cells cannot be moved, as rectangular partitions are needed (see Section 2.2). This is less flexible and could lead to possible oscillations in the load (see Section 2.9).

The processor computation time of the imbalanced loop can be used to calculate the overall column times (or the row times if the Left/Right limits were staggered in Figure 2.10) in order to find a 'weighting' (time per column of cells). Using these weights, the columns of cells can be distributed evenly, giving more columns to those with smaller weights, thus reducing the workload on those with heavier weights.

It is assumed that each cell on a processor takes the same amount of time to compute, as it would be impractical to actually code and time each individual cell rather than the whole workload on that processor. This assumption is actually true in the case of processor imbalance, where there is no physical phenomena, but it would obviously be difficult to distinguish which cells need more processing power than others when physical imbalance is suspected. This issue is covered in more detail in Section 3.5.5.

The new Up/Down limits for each column of processors can now be determined having balanced the load in the Non-Staggered Dimension, whereby the observed timings are adjusted to take into consideration the balance in the previous dimension. Each column of processors is balanced individually, using the weights (time per row of cells) for each processor in the same way as previously mentioned (where the weight is no longer for a column of processors but for a single row on a processor within a column of processors). This means that the Up/Down limits for the processors in each of the different columns of processors can have different values, allowing staggering to occur.

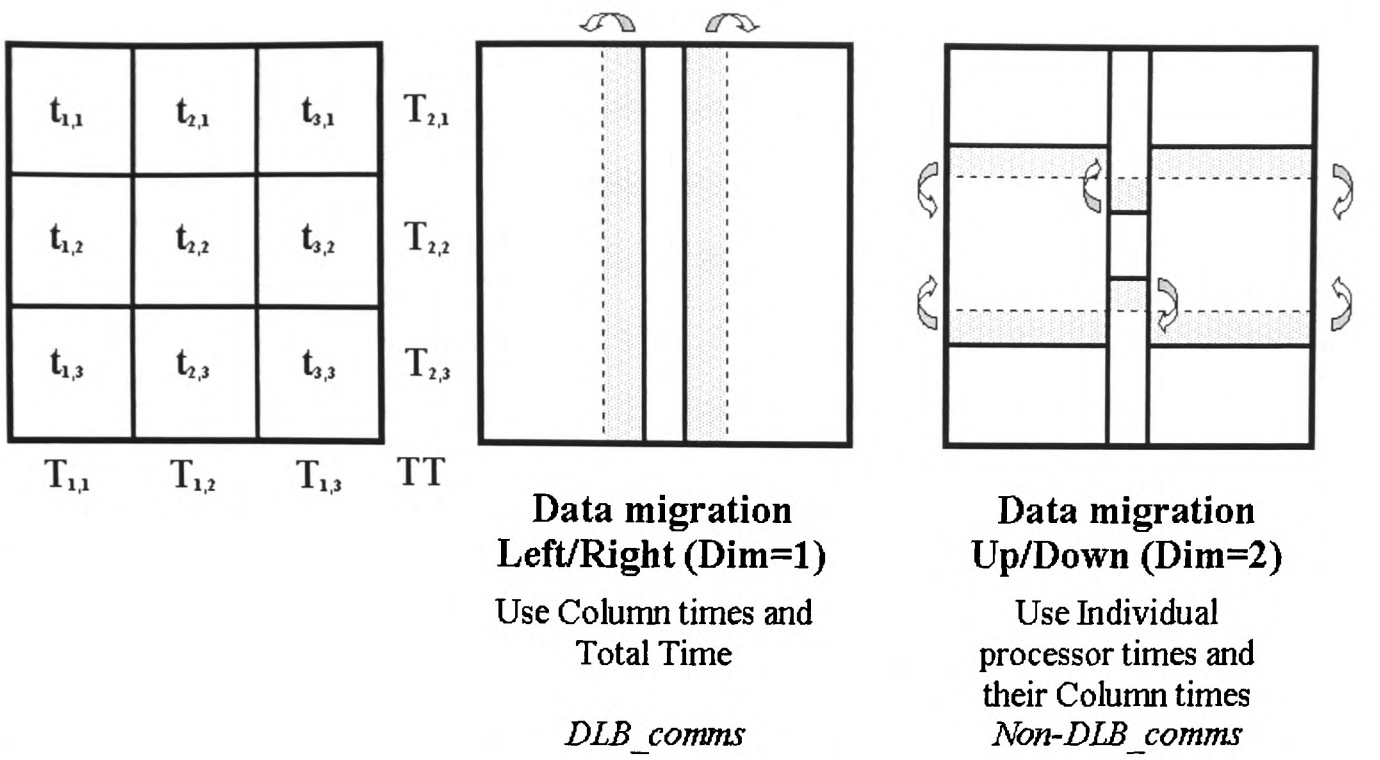


Figure 2.10: Data migration for a two-dimensional processor topology, with global Left/Right processor partition range limits and staggered Up/Down processor partition range limits

Imbalance could be due to the physical imbalance or processor imbalance, or both, and this affects the way in which the timings are adjusted. In both situations, if a processor loses some of its load then its timing will be reduced proportionally. If a processor gains some workload from a neighbouring processor, then it will either adjust its own time proportionally again if the imbalance is due to the processor imbalance, or it will adjust its own time relative to the neighbouring processor if the imbalance is due to physical imbalance. A processor will either gain an additional cell at the weight of its neighbour or at its own weight depending on the type of problem, but it may be very difficult to determine how to adjust the timings if there is both physical and processor imbalance. This issue is crucial if an effective distribution is to be obtained (Section 3.5.3).

With processor imbalance the problem is balanced and so all of the cells have the same computational load associated with them. The variation between the processors (speed and number of users/jobs) implies that the cells are processed at different rates (fast or slow), and so the weight of the cell is not being transferred across, i.e. a fast processor would not process any additional cells gained from a slow processor at the slower rate.

With physical imbalance the problem is that each cell can have a different computational load associated with it, where there is no variation between the

processors. In this instance when a lightly loaded processor gains additional cells from a heavily loaded processor the load associated with those cells is transferred across where the cells still have the same amount of computational work associated with them, but the work is now handled by another processor.

There are a number of constraints to comply with when calculating the new partition range limits, each of which must be considered when balancing the load in each of the partitioned dimensions. In order to utilise the available resources, and to preserve the authenticity of the processor topology, each processor must contain a minimum number of cells. This is necessary since neighbouring processors still need to be able to retrieve data into their halo region, which has been assigned on another processor (see Figure A.15). Therefore the minimum width on each processor is essentially equivalent to the halo width (for each dimension), which will allow data to flow from processor to processor (acting as a filter). If a minimum width were not imposed on each processor, then it would be possible for a situation to arise where a processor has no work to process whatsoever. The minimum number of slabs (MINSLABS) specified by the user within CAPTools needs to be satisfied, otherwise it may be possible that the halo region is updated incorrectly.

Memory reduction also acts as a constraint, this time on the maximum number of cells owned. After memory reduction each processor owns a subset of the original data (as there is no need to store the whole data array), and so they own a limited amount of data space in which to place any gained data. Therefore the new load is constrained by the size of the memory.

Another constraint is the goal set in Section 2.1, which insists on attempting to minimise movement of data, where data can only be gained from a neighbouring processor in any one redistribution, giving the impression of a gradual movement of data rather than bulk movement. This means that when gaining cells from a neighbour, the number of cells on the neighbour must not fall below its minimum amount, as this would conflict with the above constraint. It would also mean that more communications would be necessary, as its neighbour's neighbour would need to pass its data along too.

2.8 Implementing The New Distribution

The new distribution cannot be implemented correctly until each processor owns the data defined by its new limits, hence the need for load migration.

As mentioned earlier, load migration is a fundamental component of this DLB strategy, as it is where data is transferred from one processor to another, via a set of communication calls to construct a new partition. When a new load distribution is established it is necessary to ensure the correct processor ownership of data, so that each processor owns the current values of all the data defined by its new processor partition range limits. If the new limits were used in subsequent code without the data first being migrated then the processors would be using incorrect or uninitialised data in their calculations. Additionally some data values would not be known, such as data in the halo region.

Load migration needs to be efficient, particularly since a significantly large number of arrays representing geometric, physical and chemical properties, may need to be migrated (often 100+), which could prove costly in execution time. Obtaining an efficient load migration is essential so that this stage does not overshadow the saving in execution time achieved by employing the new partition. This requires that the migration stage should be fast, only moving a minimum amount of data, and using few communications as possible, operating in parallel, if possible. An important requirement of any such algorithm is that it typically allows the vast majority of program data to remain where it is and only moves a small proportion in order to set up the new partition, as specified in goal 13 in Section 2.1.

To avoid communication latencies and unnecessary data movement, it would be ideal to use a minimum number of communication calls to migrate the load, which is why the manner in which data migration occurs needs to be noted. If the load is migrated using all of the newly calculated partition range limits then this essentially would mean that the load would need to be communicated either directly with the new owning processor, or through a number of communications which does not comply with the objective set above. Solely using the old partition range limits to migrate the load would not be suitable either, because some data would not be transferred as only data within the old processor partition range

limits are transferred. However, if the load is migrated in one dimension using the old partition range limits, and then migrated in the next dimension using the new limits of the previously migrated dimension, then all data is transferred onto the owning processor without the need for 'diagonal' communications. Although there is a specific order in which the new limits are calculated, the order in which the data is migrated is not significant. It makes no difference whether the data is migrated Left/Right and then Up/Down, or vice versa, so long as the limits of recently migrated dimensions are used when migrating subsequent dimensions.

The order in which the load is migrated is not considered a high priority so long as the data is migrated correctly with minimum movement. It has been decided that data in the Non-Staggered Dimensions shall be migrated first, followed by the Staggered Dimension, simply because this is the order in which the partition range limits are calculated.

Data is first migrated in a particular direction, using the values of the specified processor partition range limits, and then migrated in the other direction, using the newly specified limits. In Figure 2.10, it can be seen that the load is first migrated in the Left/Right direction (Non-Staggered Dimension), communicating within the old Up/Down limits, where the new Left/Right limits are internally compared to the old Left/Right limits. Then the load is migrated in the Up/Down direction (Staggered Dimension), internally comparing the new Up/Down limits to the old Up/Down limits, and communicating within the new Left/Right limits.

As mentioned above, the load migration of a particular variable is essentially a collection of communication calls that transfer data to neighbouring processors, this can appear obtrusive, and so a single generic call to do this would be more advisable if some attempt is made to minimise code changes. The direction, start address, and amount of data to be migrated, will differ for each variable in each redistribution, where the migration message may additionally be dissected between several processors when migrating in a Non-Staggered Dimension.

Processors will only need to communicate with immediate neighbours when migrating data for the first time, since global partition range limits are still in use. Once the limits have been staggered then processors will be communicating with several neighbours in a Non-Staggered Dimension. Therefore two migration calls are needed, whereby the parameters of the call are

internally used to determine the communication call used to migrate the load in either the Staggered or Non-Staggered Dimension. The underlying operations of these migration calls are similar to the requirements of the new DLB communications (in Section 2.5.3) that are needed to communicate over the staggered limits. How much data to communicate, and to whom to communicate with, is determined internally by comparing the processor partition range limits.

Using this strategy in which the Non-Staggered Dimensions are migrated first, the processor partition range limits of the migrated dimension must be reassigned on each processor before migrating the load in the following dimensions, for use in the subsequent code (and internally for use in the utility calls).

After migration each processor owns the data defined by their new partition range limits, however, they may need to use data in the halo region that is owned by neighbouring processors. After the load migration stage (in which processor ownership is ensured), an overlap Exchange communication, say, will involve current data. The problem arises when halo data that was updated before changing the distribution is used after redistribution, as the current value has not been migrated. This suggests that some overlap communications that occur before redistribution may need to be duplicated after the load migration stage to ensure that valid halo data exists before continuing.

2.9 Load Oscillation

Models of when to redistribute the workload between processors, and how much to migrate, are based on several assumptions (Sections 2.7.2.2 and 2.7.4). Obviously these assumptions are often not correct, and so damping (underestimating) is used to avoid load oscillation at the cost of a subsequent redistribution. For example, the assumption that the increase in load imbalance is linear is not necessarily true for processor imbalance, as the load imbalance is approximately constant, varying in the number of users/jobs rather than speed. This is also true for the case in which there is a constant level of physical imbalance, in which the geometry of the problem does not change during

execution. This assumption is only rarely true for the case in which there is a varying degree of physical imbalance, where the physical characteristics of the problem change continuously throughout execution (with a growing level of load imbalance). This assumption implies that with static imbalance (those cases just mentioned) the linear increase encourages redistribution so that the imbalance is removed.

One of the other assumptions made was that redistribution removes all of the load imbalance, which is rare. The granularity of the problem itself prevents a perfect balance being attained, since the single cells of a structured mesh cannot be moved, only a row or column, etc can be moved. We can only assume that the load imbalance is removed completely, otherwise we have to try and estimate exactly how much remains after redistribution, complicating the issue of load balancing further.

Another assumption was that all cells on a processor (or set of processors) have the same weight, which is not always true. With processor imbalance this assumption may be false, as the cell weight when calculating the processor partition range limits in a Non-Staggered Dimension will be different for processors of varying speed (and number of jobs/users). For example, when calculating the new workload on a set of processors in the Left/Right direction, the processors are grouped into columns of processors. It is assumed that the cells on each processor in the group (column) have the same weight, where this is often not true. This assumption is also not true when any physical imbalance is present, since the imbalance may be due to just one or two heavy cells or a variation over all cells.

The user may want to prevent load oscillations from occurring, where cells are being moved to and fro, so that time is not wasted migrating the same cells from one redistribution to the next. In such cases it may be desirable to set some constraint on the minimum amount of cells that can be moved in subsequent redistributions, either in total or in a given dimension, implying that load redistribution should only occur if enough cells are to be moved.

With processor imbalance each cell has the same computational weight, as there is the same amount of work associated with each cell. This implies that the performance of the parallel code may not improve dramatically by the movement of a single cell (or even a few cells). The ideal number of cells may not be

migrated due to the fact that single cells cannot be moved, and so either too many or too few cells are migrated instead, which can oscillate between redistributions. In this instance setting a constraint on the minimum amount of cells to be moved could possibly be used to avoid such oscillations.

With physical imbalance however, the situation is different due to the fact that each cell has a differing amount of computational work associated with it due to physical phenomena. This implies that the performance of the parallel code may improve dramatically by the movement of a single cell (or even a few cells). In this situation load oscillations could be due to either the physical phenomena but it could still be due to the granularity of the structured mesh code, where the ideal number of cells may not be moved. The load can oscillate naturally in this situation, which makes it difficult to say that redistribution should be delayed if not enough cells are migrated, as the quantity is no longer an issue here, as potentially a single cell can influence the load balance. If the load were not migrated due to the fact that not enough cells were to be moved then the load imbalance could continue at its present rate.

With physical imbalance a cell on a processor may have a lot of work associated with it, and so it is calculated that some cells on this processor will be moved. However, this actual heavy cell may be in the centre of the processor's load, and so is not moved, and so its timing does not reduce that much. After a couple of redistributions this heavy cell may be taken off, but then its burden is simply placed onto another processor. The heavy cell may be transferred back and forth between processors. Ideally, it would be desirable for the heavy cell to end up on its own on a processor (reducing the maximum processor time).

In this situation, with processor imbalance more cells are moved each time the load is redistributed, since they appear 'cheaper' after moving. With physical imbalance fewer cells are moved each time, as they can still appear 'expensive' after moving. There is therefore a higher chance of load oscillation with processor imbalance than there is with physical imbalance. This is illustrated in Figure 2.11, where the final distribution is dependent on the location of the heaviest cell.

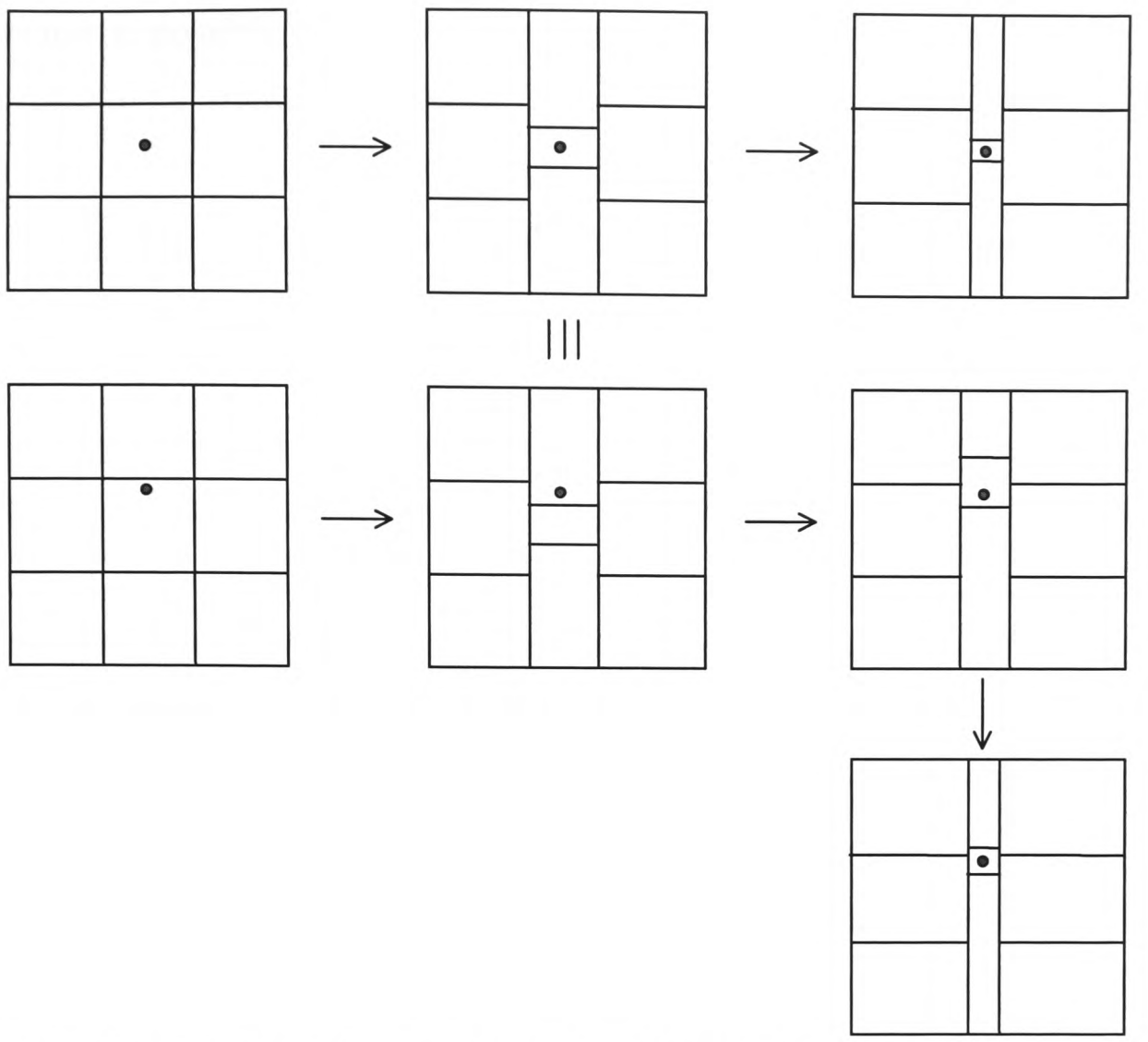


Figure 2.11: Example illustrating how the load would move given the location of the heaviest cell when assuming physical imbalance.

2.10 Goals Of The DLB Staggered Limit Strategy

The need for DLB has been demonstrated in the examples given in Section 1.11, where speed and accuracy are of great importance. To improve the efficiency of an imbalanced code an attempt must be made to minimise the maximum processor time (consequently reducing the idle time), which can only be achieved by redistributing the load when the level of load imbalance becomes significant, enabling processors to finish computing in the same amount of time. A major cost of a DLB algorithm is the time to calculate and redistribute the program data, especially since a large number of program arrays may need to be migrated to satisfy a new partition. If this migration is too expensive, the improvements achieved by the new partition may be offset by the redistribution cost, and so the profitability of a load redistribution is measured by taking both the level of idle

time and the cost of redistribution into account. The cost of migration must be kept as low as possible to the effectiveness of the selected DLB strategy.

Additionally, the user should still be able to recognise and maintain the DLB version of the code to allow continued maintenance and optimisation, and so utilities should be developed to avoid major changes to the code.

2.11 Summary

When manually implementing a DLB strategy within an application code the user will always consider the effort needed in order to do so, which may have some side-effects. If the DLB strategy is too complex to implement manually within a code, or several codes, then a simpler strategy may be employed, with a less qualitative balance, discouraging further usage of this strategy due to the lack of benefits. The user must determine whether the DLB strategy is relatively easy to implement within several codes, and whether the attainable balance will be significant.

Much of the current research (see Section 1.14) concentrates on one specific area (e.g. what to migrate), or on a specific application code, or on specific machines. The proposed DLB Staggered Limit Strategy incorporates some of the aspects from the current research, but more importantly, it ties them together and allows for the automatic implementation of this strategy within a CAPTools generated parallel code.

This Chapter has examined several DLB strategies, where it was decided that the DLB Staggered Limit Strategy that uses coincidental processor partition range limits in all but one of the partitioned dimensions would be the most appropriate choice for automating within CAPTools. The Staggered Dimension was defined to be the partitioned dimension containing the local (non-coincidental) limits, whereas the Non-Staggered Dimensions were defined to be those partitioned dimensions containing the global (coincidental) limits. The communication structure of the DLB Staggered Limit Strategy is not as complex as Case 2 (Figure 2.2), where non-coincidental limits were used in all partitioned dimensions. Additionally, due to the flexibility provided by the staggered limits, a

better balance is attainable than Case 1 (Figure 2.2) in which coincidental limits were used in all partitioned dimensions. However, it was demonstrated that the communication structure would need to be altered when using the DLB Staggered Limit Strategy, as it was now possible for a processor to communicate with potentially several neighbouring processors in a Non-Staggered Dimension, rather than just its immediate neighbour.

This Chapter also discussed the practicalities of DLB, such as the location in the user's code at which to redistribute the workload, how often to redistribute the workload, and calculating and implementing the new workload. The load imbalance is usually contained in a loop (time-step, iteration and solver), where the location at which the load is redistributed in this loop is only important in terms of automation (see Section 5.7.2). It was decided that a model of computation would be used to determine when to redistribute the workload, since balancing infrequently would lead to the idle time becoming significant, and balancing too often would lead to the redistribution time becoming significant. The model of computation decides when it is plausible to redistribute the workload based on the level of load imbalance and the cost of redistributing the workload.

The effect of processor and physical imbalance was highlighted when discussing the calculation of the new workload. It was decided that the assumption that every cell on a processor is processed at the same rate (weight) would be used for simplicity, however although this assumption can be true for processor imbalance it is untrue for physical imbalance (but necessary nevertheless). Each partitioned dimension would need to be processed separately, where the processor computation timings would have to be adjusted before processing subsequent dimensions to account for the 'balance' already obtained. Cells would be lost at a processor's own weight for both processor and physical imbalance. Cells would also be gained at a processor's own weight for processor imbalance, but would be gained at a neighbouring processor's weight with physical imbalance.

Having calculated the new workload, the new distribution can be implemented by ensuring processor ownership of that distribution. How much data to move and the communication direction need to be established, where the data is communicated with the new owners, involving immediate neighbours in the Staggered Dimension, but involving several neighbouring processors in the

Non-Staggered Dimensions due to the staggered limits. The halo region also needs to be updated for some arrays at this stage, allowing the parallel code to execute correctly with the use of up-to-date halo data.

Generic utilities for the DLB Staggered Limit Strategy shall be described in the following Chapter, where its manual implementation within a CAPTools generated parallel code is reviewed in Chapter 4 and its automation is discussed in Chapter 5.

Chapter 3 Generic Dynamic Load Balancing Utilities

Chapter 1 discussed the need for DLB, where the resultant DLB Staggered Limit Strategy was devised in Chapter 2. In the context of CAPTools, within which this strategy is to be automated, all but one of the partitioned dimensions uses coincidental processor partition range limits, where the remaining dimension uses non-coincidental (staggered) processor partition range limits. The following generic utilities were devised in order to conform to the goals set in Section 2.1, where the main objectives are to promote functionality for the DLB strategy efficiently, maintaining the logical process topology whilst trying to minimise the changes to the user's code. As few parameters as possible are used in these generic utilities, keeping much of the data internal, reducing the amount of information the user needs to know in order to implement DLB within their code.

3.1 Generic Utilities

A utility is a procedure, or function, which is used to perform some task, which should not be written specifically with a particular type of problem in mind. The task must be applicable to a wide range of application codes and not just a select few (see Section 1.8), which is why all of the utilities need to be generic. Most of the utilities discussed in this Chapter operate in bytes, enabling the utilities to be used for any data type, reinforcing their generic function.

Inserting a single call statement into the code, rather than the precise code of the utility, keeps the code neat and simple, as it remains readable and uncluttered. The user need not know the exact underlying operations of the utility, but only need know that a specific task is performed when the utility is executed, ensuring that the original code can still be maintained and optimised without any need to change the utility.

3.2 Initialising DLB Mode

In order to execute the parallel code in DLB mode certain DLB variables need to be set up, enabling the DLB utilities to operate correctly. A utility is needed to set up the processor connectivity so that each processor knows who its neighbouring processors are in every direction, for the processor topology specified at runtime (see Section A.2). Additionally, each processor needs to know the processor partition range limits of all of their neighbours as well as knowing their own limits, and so these need to be stored. For example, when communicating over non-coincidental limits the staggered limits of those processors involved need to be compared, and when calculating the new partition range limits, the limits of adjacent processors need to be known as well as identifying the adjacent processor.

3.2.1 Store Processor Neighbours

Each processor needs to know which neighbours are contained in the adjacent 'block' of processors (as discussed in Section 2.5.2). The number of processors in each block is equivalent to the number of processors specified at runtime for the Staggered Dimension, and the adjacent blocks are contained within the dimensions orthogonal to the Staggered Dimension (Figure 2.3).

Using the 2D-grid shown in Figure 3.1 as an example, Table 3.1 contains information relating to the neighbouring processors in every direction for each processor in which the Up/Down limits have been staggered. This information may alternatively be referred to as the communication structure, as it indicates which processors can communicate with one another. For example, in the original communication structure Processor 5 would only have been communicating with its immediate neighbours 6 and 4 in the Left/Right direction and 2 and 8 in the Up/Down direction. Using the new communication structure Processor 5 can still communicate with Processors 2 and 8 in the Up/Down direction, but now it can potentially communicate with Processors 1, 6 and 7 in the Left direction, and Processors 3, 4 and 9 in the Right direction. Similarly when using the 3D grid

shown in Figure 2.3, Processor 14 would no longer be communicating with just Processors 15 and 13 in the Left/Right direction, 11 and 17 in Up/Down direction, and 5 and 23 in the Back/Forth direction. Processor 14 would also be able to potentially communicate with 8 other processors (2, 8, 10, 12, 16, 18, 20, and 26), however it will not need to communicate with 'diagonal' processors, such as Processors 1, 6, and 7, for instance. The data is communicated vertically and horizontally (and vice versa), avoiding diagonal communication (Section A.3.3). Note that when a processor has no neighbour in a given direction (indicated by a 0) then its cyclic neighbours are shown in brackets. For example, Processor 1 has no neighbours to its Left, although when a torus type topology (Section A.2) is specified then Processor 1 may potentially communicate with Processors 3, 4 and 9 to its Left.

1	2	3
6	5	4
7	8	9

Figure 3.1: 2D grid in which the Up/Down processor partition range limits may be staggered.

Processor	Left	Right	Up	Down
1	0/(3, 4, 9)	2, 5, 8	0/(7)	6
2	1, 6, 7	3, 4, 9	0/(8)	5
3	2, 5, 8	0/(1, 6, 7)	0/(9)	4
4	2, 5, 8	0/(1, 6, 7)	3	9
5	1, 6, 7	3, 4, 9	2	8
6	0/(3, 4, 9)	2, 5, 8	1	7
7	0/(3, 4, 9)	2, 5, 8	6	0/(1)
8	1, 6, 7	3, 4, 9	5	0/(2)
9	2, 5, 8	0/(1, 6, 7)	4	0/(3)

Table 3.1: Shows the DLB communication structure for the example grid in Figure 3.1, where the Staggered Dimension contains the Up/Down processor partition range limits. The neighbouring processors are shown for each direction, where a 0 indicates no neighbours, and the cyclic neighbours are shown in brackets (where different to the ordinary neighbours).

Information pertaining to the number of partitioned dimensions, and the number of processors in each of these, along with the processor number and position, need to be known in order to set up this 'neighbouring' information. The neighbouring information is stored in the array ALLNEIGHBOURS(Neighbour_Number,Direction), which stores the processor

number of every neighbour in each direction. Figure 3.2 illustrates how this information is stored for the examples shown in Figure 3.1 and Figure 2.3. Note that the directions Left, Right, Up, Down, Back, and Forth, can all be specified within the code using –1, -2, -3, -4, -5, and –6 (Section A.3.3.1). For example, the first Left neighbour of Processor 5 in Figure 3.1 is Processor 1, the second neighbour to its Left is Processor 6, and its third Left neighbour is Processor 7.

Similarly, CYCNEIGHBOURS(Neighbour_Number,Direction) stores the cyclic neighbours of each processor, where a 0 is used to indicate that there is no neighbour in the given direction.

For Processor 5 in the 2D grid shown in Figure 3.1:

ALLNEIGHBOURS(1,-1)=1	Left		Right	ALLNEIGHBOURS(1,-2)=3
ALLNEIGHBOURS(2,-1)=6				ALLNEIGHBOURS(2,-2)=4
ALLNEIGHBOURS(3,-1)=7				ALLNEIGHBOURS(3,-2)=9
ALLNEIGHBOURS(1,-3)=2	Up		Down	ALLNEIGHBOURS(1,-4)=8

For Processor 14 in the 3D grid shown in Figure 2.3:

ALLNEIGHBOURS(1,-1)=10	Left		Right	ALLNEIGHBOURS(1,-2)=12
ALLNEIGHBOURS(2,-1)=15				ALLNEIGHBOURS(2,-2)=13
ALLNEIGHBOURS(3,-1)=16				ALLNEIGHBOURS(3,-2)=18
ALLNEIGHBOURS(1,-3)=11	Up		Down	ALLNEIGHBOURS(1,-4)=17
ALLNEIGHBOURS(1,-5)=2	Back		Forth	ALLNEIGHBOURS(1,-6)=20
ALLNEIGHBOURS(2,-5)=5				ALLNEIGHBOURS(2,-6)=23
ALLNEIGHBOURS(3,-5)=8				ALLNEIGHBOURS(3,-6)=26

Figure 3.2: Examples of what is stored in ALLNEIGHBOURS, for Figure 3.1 and Figure 2.3.

A call to CAP_DLB_SETALLNEIGHBOURS is used to set up the neighbouring processors for every processor when DLB has been selected, where no parameters are needed since internal CAPLib variables are used. This call is similar to CAP_INIT (Section A.2) in which the parallel parameters are set up before executing any parallel statements. It needs to be placed above any DLB code (such as DLB communication calls) to ensure correct implementation, and so this call should ideally be placed as high up in the code as possible, preferably immediately after CAP_INIT.

3.2.2 Store Processor Partition Range Limits Of Neighbours

The lower and upper processor partition range limits of every processor need to be stored for each dimension, which have been set up in CAP_SETUPPART or CAP_SETUPDPART, depending on the number of partitioned dimensions (see Section A.2). The processor axes also need to be passed in so that the different partition range limits can be stored under the correct partitioned dimension. On the first pass CAP_LOW and CAP_HIGH were generated, and so these should be stored in the first processor axes, and likewise for CAP2_LOW and CAP2_HIGH, which should be stored under the second processor axes.

The array CAP_DLB_PROCLIMITS(Limit_Index,Processor_Number) is used to store the processor partition range limits, which are passed into the utility. Figure 3.3 shows the call statements to set up the limits of the processors in which LOW, HIGH and IAXES are passed into the actual utility shown in Figure 3.4. Each processor is then able to extract the value of a neighbouring processor's partition range limits after execution of these CAP_DLB_SETUPLIMITS calls.

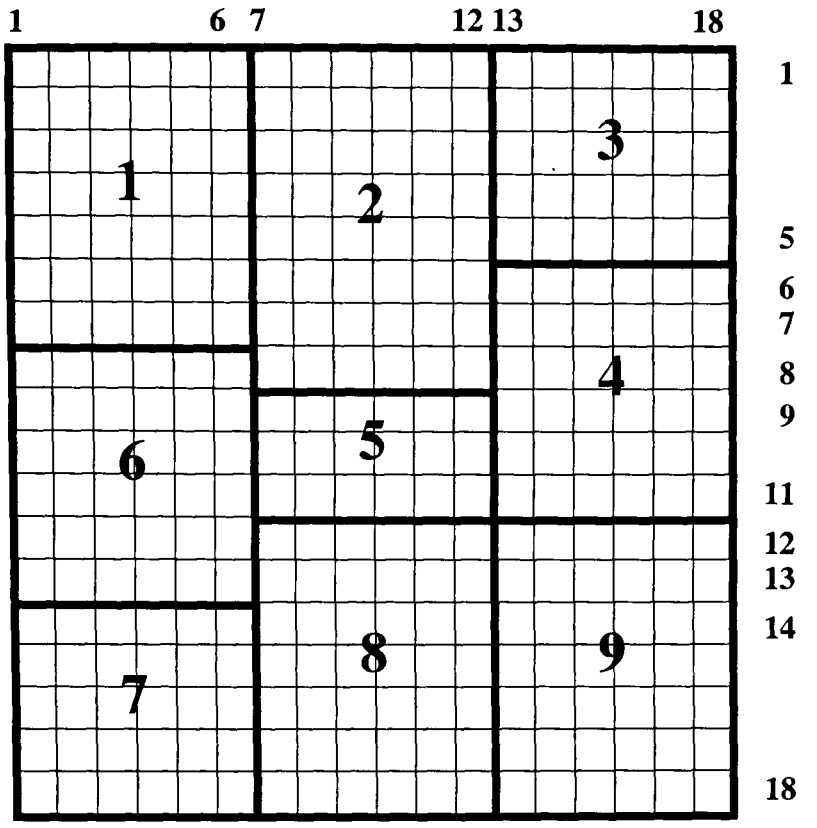
```
C       cap1_low/cap1_high, and cap2_low/cap2_high, already
C       set-up using CAP_SETUPDPART
        CALL CAP_DLB_SETUPLIMITS(CAP1_LOW,CAP1_HIGH,1)
        CALL CAP_DLB_SETUPLIMITS(CAP2_LOW,CAP2_HIGH,2)
```

Figure 3.3: Call statements used to internally set up the processor partition range limits of all processors.

```
      SUBROUTINE CAP_DLB_SETUPLIMITS(LOW,HIGH,IAXES)
C     Declarations
      INTEGER CAP_DLB_PROCLIMITS(MAXINDEXNO,MAXPROCS)


C     Set up low and high in a particular iaxes (direction)
C     i.e.: low  = (iaxes*2)-1
C           high = (iaxes*2)
C     Ex       for I (iaxes 1)   -> low=1 (1*2-1), high=2 (1*2)
C              for J (iaxes 2)   -> low=3 (2*2-1), high=4 (2*2)
C              for K (iaxes 3)   -> low=5 (3*2-1), high=6 (3*2)

C     Set index of array containing low and high limits
      INDEX_LOW=(IAXES*2)-1
      INDEX_HIGH=(IAXES*2)

C     Store the lower limit of this processor for the given iaxes
      CAP_DLB_PROCLIMITS(INDEX_LOW,CAP_PROCNUM)=LOW

C     Store the higher limit of this processor for the given iaxes
      CAP_DLB_PROCLIMITS(INDEX_HIGH,CAP_PROCNUM)=HIGH

C     Broadcast processor limits to all other processors
      DO I=1,CAP_NPROC
C        Do not send to self
         OWNER=.FALSE.
         IF( I.EQ.CAP_PROCNUM ) OWNER=.TRUE.
C        Allow all other processors to know the array contents of Processor I
         CALL CAP_BROADCAST(CAP_DLB_PROCLIMITS((2*IAXES)-1,I),
     2,1,OWNER)
      END DO
```

Figure 3.4: Code used to store the processor partition range limits for each processor in the specified dimension.

Using the 2D grid in Figure 3.1 as a basis for illustration, the contents of CAP_DLB_PROCLIMITS after a redistribution are shown in Figure 3.5 alongside the redistributed load. Since CAP_DLB_PROCLIMITS is stored on every processor, each processor knows that Processor 5's Left processor partition range limit is 7, its Right limit is 12, its Up limit is 9, and its Down limit is 11.



CAP_DLB_PROCLIMITS(1,1)=1
CAP_DLB_PROCLIMITS(2,1)=6
CAP_DLB_PROCLIMITS(3,1)=1
CAP_DLB_PROCLIMITS(4,1)=7

CAP_DLB_PROCLIMITS(1,2)=7
CAP_DLB_PROCLIMITS(2,2)=12
CAP_DLB_PROCLIMITS(3,2)=1
CAP_DLB_PROCLIMITS(4,2)=8

CAP_DLB_PROCLIMITS(1,3)=13
CAP_DLB_PROCLIMITS(2,3)=18
CAP_DLB_PROCLIMITS(3,3)=1
CAP_DLB_PROCLIMITS(4,3)=5

CAP_DLB_PROCLIMITS(1,4)=13
CAP_DLB_PROCLIMITS(2,4)=18
CAP_DLB_PROCLIMITS(3,4)=6
CAP_DLB_PROCLIMITS(4,4)=11

CAP_DLB_PROCLIMITS(1,5)=7
CAP_DLB_PROCLIMITS(2,5)=12
CAP_DLB_PROCLIMITS(3,5)=9
CAP_DLB_PROCLIMITS(4,5)=11

CAP_DLB_PROCLIMITS(1,6)=1
CAP_DLB_PROCLIMITS(2,6)=6
CAP_DLB_PROCLIMITS(3,6)=8
CAP_DLB_PROCLIMITS(4,6)=13

CAP_DLB_PROCLIMITS(1,7)=1
CAP_DLB_PROCLIMITS(2,7)=6
CAP_DLB_PROCLIMITS(3,7)=14
CAP_DLB_PROCLIMITS(4,7)=18

CAP_DLB_PROCLIMITS(1,8)=7
CAP_DLB_PROCLIMITS(2,8)=12
CAP_DLB_PROCLIMITS(3,8)=12
CAP_DLB_PROCLIMITS(4,8)=18

CAP_DLB_PROCLIMITS(1,9)=13
CAP_DLB_PROCLIMITS(2,9)=18
CAP_DLB_PROCLIMITS(3,9)=12
CAP_DLB_PROCLIMITS(4,9)=18

Figure 3.5: Example in which the processor partition range limits are staggered in the Up/Down direction (second partitioned dimension). Also shown are the contents of CAP_DLB_PROCLIMITS, known by all processors, indicating the partition range limits of each processor.

3.3 Communicating Across Non-Coincidental Processor Partition Range Limits

After redistribution using the DLB Staggered Limit Strategy, processors may have to communicate over non-coincidental partition range limits. Existing communications within CAPTools are not capable of handling communications over the staggered limits, as originally each processor only had to communicate with their immediate neighbours. Only communications in a Non-Staggered Dimension (dimension containing coincidental limits) will be affected by the staggered limits, as processors still only need communicate with immediate neighbours in the Staggered Dimension. Where processors were originally communicating with one processor, they may now have to communicate with several neighbours. For example, in Figure 3.5 Processor 6 will still only need to communicate with Processor 1 in the Up direction, but will now have to communicate with Processors 2, 5, and 8, when communicating to its Right. In this example, although Processor 9 only needs to communicate with its immediate neighbour in the Non-Staggered Dimension (Processor 8), this may not always be the case due to load redistribution. If the load is redistributed again, then Processor 9 may also have to communicate with Processor 5, implying the necessity to store all potential processors in order to dynamically determine who to communicate with.

3.3.1 Splitting The Communication Message

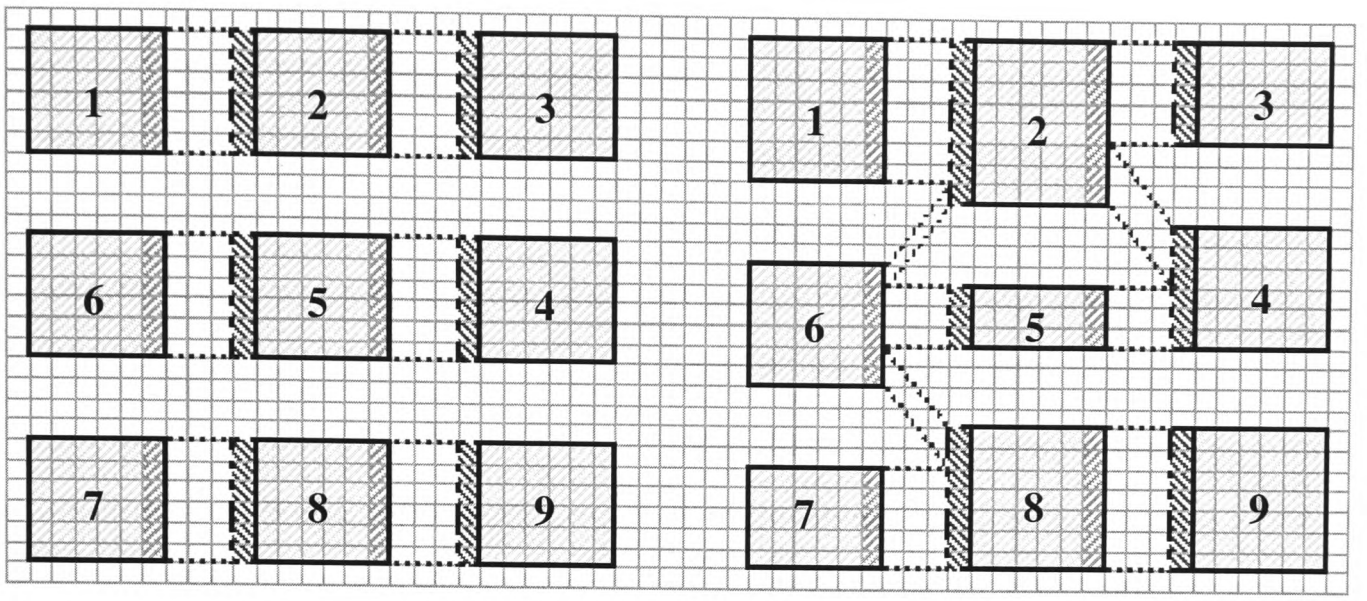
Parallel structured mesh codes generated using CAPTools use the abstraction of a communication direction (or processor identifier, PID), which can be exploited in the DLB communications. Determining whom to communicate with, and how much to communicate can be achieved by dissecting the original communication message into several communication messages involving the appropriate neighbour in the specified direction. A processor should only ever need to communicate with 'intersecting' (overlapping) processors, whose 'communication

message' intersect in the Staggered Dimension. The neighbouring processors can be obtained from ALLNEIGHBOURS (Section 3.2). If the neighbour is equal to 0 then no neighbours exists in the specified direction, and so there is no need to continue with this communication call.

The example code shown in Figure 3.6 can be used to demonstrate the simplest of cases, where each processor assigns data between their processor partition range limits that is then needed to update the Left halo region on a neighbouring processor. The graphical illustration demonstrates the communication update on the original (non-DLB) distribution and on a staggered distribution (that shown in Figure 3.5). Processor 2 originally had to receive all of its halo data (T(CAP1_LOW-1,CAP2_LOW:CAP2_HIGH)) from Processor 1, but with the staggered distribution it now has to receive its halo data from Processor 1 and 6. A table indicating the data in the Staggered Dimension that each processor needs to receive into from their Left neighbours is also given in Figure 3.6. Processor 2 needs to receive cells T(6,1:8) in total from its Left, which means receiving T(6,1:7) from Processor 1, and receiving T(6,8) from Processor 6. Additionally, a table indicating the core data in the Staggered Dimension that each processor needs to send to their Right neighbours is given. For example, Processor 1 needs to send T(6,1:7) to Processor 2, which corresponds to the receive set in the receive table. Processor 6 needs to send T(6,8) to Processor 2, T(6,9:11) to Processor 5 and T(6,12:13) to Processor 8. Only the staggered limits need to be compared, since the communication message is being dissected in the Staggered Dimension which affects the communication.

```
DO J=CAP2_LOW,CAP2_HIGH
   DO I=CAP1_LOW,CAP1_HIGH
      T(I,J)=...
   END DO
END DO
```

```
DO J=CAP2_LOW,CAP2_HIGH
   DO I=CAP1_LOW,CAP1_HIGH
      V(I,J)=T(I-1,J)
   END DO
END DO
```



RECEIVE into the lower halo region from the Left

Receiving Processor:	Neighbours:		
P2(1:8)	P1(1:7)	P6(8:8)	P7(0)
P3(1:5)	P2(1:5)	P5(0)	P8(0)
P4(6:11)	P2(6:8)	P5(9:11)	P8(0)
P5(9:11)	P1(0)	P6(9:11)	P7(0)
P8(12:18)	P1(0)	P6(12:13)	P7(14:18)
P9(12:18)	P2(0)	P5(0)	P8(12:18)

SEND the upper core region to the Right

Sending Processor:	Neighbours:		
P1(1:7)	P2(1:7)	P5(0)	P8(0)
P2(1:8)	P3(1:5)	P4(6:8)	P9(0)
P5(9:11)	P3(0)	P4(9:11)	P9(0)
P6(8:13)	P2(8:8)	P5(9:11)	P8(12:13)
P7(14:18)	P2(0)	P5(0)	P8(14:18)
P8(12:18)	P3(0)	P4(0)	P9(12:18)

Figure 3.6: Example demonstrating that the original communication message can be dissected into the intersection of the staggered processor partition range limits, where the new message starts from CAP2_LOW, and ends at CAP2_HIGH. The original communication set and the new DLB communication set are shown, along with the message range being sent and received by each processor with their neighbouring processors.

The algorithm allowing processors to communicate over non-coincidental processor partition range limits (Figure 3.7) should perform exactly the same operation as the original communication (in which there are no staggered limits).

The data is received from the processor who made the assignment, where the limits of the processors involved can be compared to see which processors need to communicate with one another. The intersection of the staggered limits dictates the new communication length (NEW_LENGTH) since most communications occur between the processor partition range limits.

Ignoring for the moment the data type being communicated, the start of the message is usually the lower limit in the Staggered Dimension (CAP2_LOW), and the end of the communication message is often the higher staggered limit (CAP2_HIGH), as most communications just involve updating the halo region. These staggered limits can be extracted from CAP_DLB_PROCLIMITS (Section 3.2.2) on each processor using SD1 and SD2 (the Staggered Dimension indices), which indicate which processor partition range limits to process. These can take the paired values of 1=Left and 2=Right, or 3=Up and 4=Down, or 5=Back and 6=Forth, etc. This utility needs to be generic as CAPTools can partition several dimensions, which means that SD1 and SD2 should not be hard coded into this utility. In Figure 3.5 for example, the Staggered Dimension is the second partitioned dimension (for the 2D processor topology) containing the Up/Down processor partition range limits, which means that SD1=3 and SD2=4. The start (L) and end (H) of the halo communication message on Processor 2 are therefore CAP_DLB_PROCLIMITS(3,2)=1 and CAP_DLB_PROCLIMITS(4,2)=8 respectively.

```
C       SD1 and SD2 are used to access CAP_DLB_PROCLIMITS
C       Where 1=Left, 2=Right, 3=Up, 4=Down, 5=Back, and 6=Forth, ...
C       SD1 is the lower index for the Staggered Dimension i.e.: 3 in 2D, 5 in 3D
C       SD2 is the higher index for the Staggered Dimension i.e.: 4 in 2D, 6 in 3D

C       Obtain the start and end location of this communication message for
C       this processor
        L=CAP_DLB_PROCLIMITS(SD1,CAP_PROCNUM)
        H=CAP_DLB_PROCLIMITS(SD2,CAP_PROCNUM)

        DO I=1,NUMBER OF NEIGHBOURS
C         Obtain neighbour i in the given direction (PID)
C         e.g.: Left=-1, Right=-2, etc
          NEIGHBOUR = ALLNEIGHBOURS(I,PID)
          IF( NEIGHBOUR.NE.0 )THEN
C           There is a neighbour in this direction – do they overlap?

C           Obtain the start and end of the communication message for
C           the neighbouring processor
            NL=CAP_DLB_PROCLIMITS(SD1,NEIGHBOUR)
            NH=CAP_DLB_PROCLIMITS(SD2,NEIGHBOUR)

C           Obtain the new start and end index in the Staggered Dimension
            LOW=MAX(L,NL)
            HIGH=MIN(H,NH)

C           Obtain the new message length – items of data
            NEW_LENGTH=HIGH-LOW+1
            IF( NEW_LENGTH.GT.0 )THEN
C             There is an intersection (overlap) with this neighbour
C             Communicate new length with this neighbour using a low-level
C             call
            END IF

          ELSE
C           No neighbours in this direction
            GOTO 10
          END IF
        END DO
10      CONTINUE
```

Figure 3.7: General code used to dissect original communication message.

The Left halo region needs to be updated on every processor for the situation given in Figure 3.6, which means that data needs to be received from the Left. To update the Left halo region on Processor 2, for example, a comparison of the staggered limits of Processor 2 against its 3 potential neighbours needs to be made (as there are 3 rows of processors). The number of potential neighbours to compare against is simply the number of processors specified at runtime for the Staggered Dimension (CAP_DNPROC(Staggered Dimension)), where the potential neighbouring processor can be identified using ALLNEIGHBOURS (Section 3.2.1) given the specified communication direction (PID). In this

example the PID=-1 (indicating a communication with a Left neighbour), and so the first Left neighbour for Processor 2 is ALLNEIGHBOURS(1,-1)=1 (Processor 1).

The staggered limits of the potential neighbouring processor can be extracted in a similar manner to those of the communicating processor (NL and NH). For example, the staggered limits of the first potential Left neighbour of Processor 2 are found to be CAP_DLB_PROCLIMITS(3,1)=1 and CAP_DLB_PROCLIMITS(4,1)=7. An intersection of the staggered limits for Processor 2 with Processor 1 can be found by calculating the difference between the maximum of the lower limits and the minimum of the higher limits. The new communication message between Processor 2 and Processor 1 will therefore start from 1 (LOW), and will end at the 7 (HIGH). If there is an intersection between the two processors (indicated by a positive difference) then a low-level communication call is set up and executed. A low-level communication will be executed on Processor 2 that will receive 7 items of data from Processor 1.

Similarly, when Processor 2's second Left neighbour (Processor 6) is processed, the new potential communication message will start from MAX(1,8) and will end at MIN(8,13). A low-level communication will be executed on Processor 2 that will receive 8-8+1=1 item of data from Processor 6 from its Left. Likewise, Processor 2 will receive 0 (MIN(1,14)-MAX(8,18)+1=-16) items of data from Processor 7, which complies with the graphical representation in Figure 3.6.

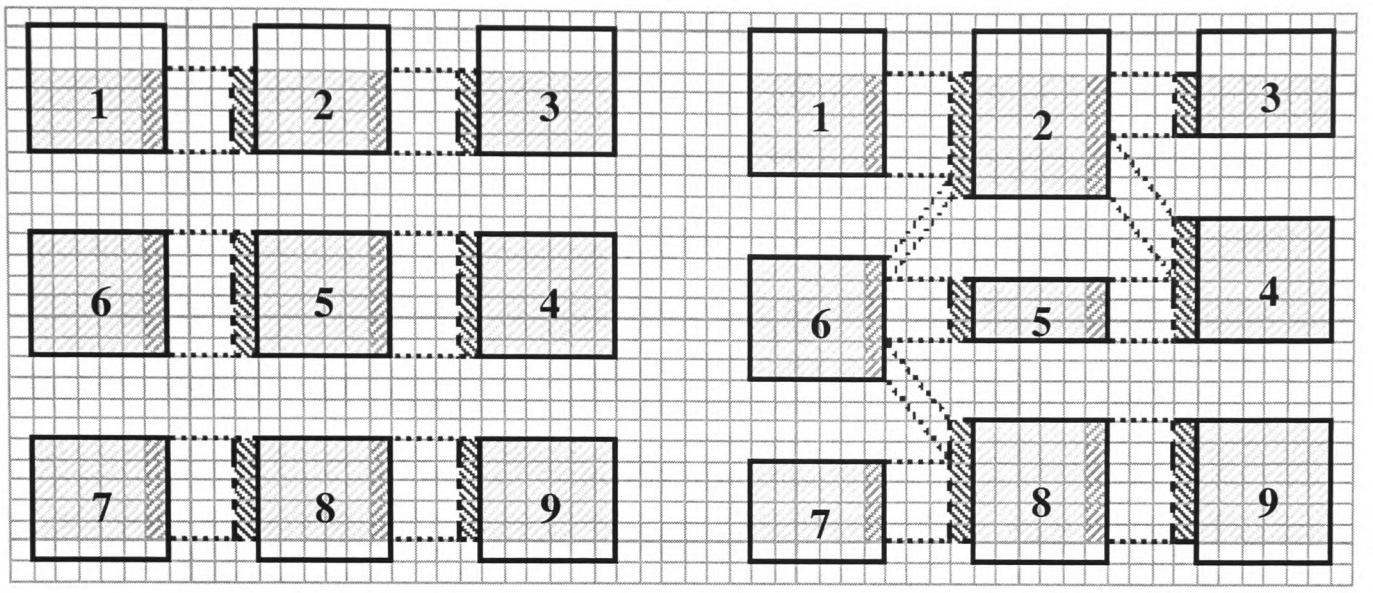
The algorithm described here (Figure 3.7) is used to dissect the original communication call (which is in a Non-Staggered Dimension) by computing the intersection of the staggered processor partition range limits, where it should be observed that every processor is evaluating their own low-level communications. Note that at this stage no additional parameters are needed in order to accomplish the operation of the original communication.

3.3.1.1 Communication Start And End

Not all communication messages start and end exactly between the processor partition range limits, as demonstrated in the example in Figure 3.8. The data on each processor is assigned between MAX(3,CAP2_LOW) and MIN(NJ-1,CAP2_HIGH) in the J dimension, where the Left halo region needs to be updated between these limits on each processor. This means that although the middle row of processors will need to use data between their staggered processor partition range limits, this is not true for the first or last row of processors. Those processors in the first row (Processors 1, 2, and 3) will need to communicate data starting from the third row (J=3), and similarly the last row of processors (Processors 6, 7, and 8) will only need to communicate data up until J=NJ-1. It would be wrong to simply compare the processor partition range limits of the communicating processor against the limits of its neighbours using the algorithm in Figure 3.7, as the communication message does not necessarily start and end at these limits. For example, when updating the Left halo region on Processor 2 in Figure 3.8, which starts from 3 and ends at 8, the new communication message with Processor 1, using the algorithm in Figure 3.7, would incorrectly start from 1 and not from 3 (although it would still end at 7). Similarly, the new communication message to update the Left halo region on the last row of processors (7, 8, and 9) should end at 17 and not 18.

```
DO J=MAX(3,CAP2_LOW),
            MIN(NJ-1,CAP2_HIGH)
  DO I=CAP1_LOW,CAP1_HIGH
    T(I,J)=...
  END DO
END DO
```

```
DO J=MAX(3,CAP2_LOW),
            MIN(NJ-1,CAP2_HIGH)
  DO I=CAP1_LOW,CAP1_HIGH
    V(I,J)=T(I-1,J)
  END DO
END DO
```



RECEIVE into the lower halo region from the Left

Receiving Processor:	Neighbours:		
P2(3:8)	P1(3:7)	P6(8:8)	P7(0)
P3(3:5)	P2(3:5)	P5(0)	P8(0)
P4(6:11)	P2(6:8)	P5(9:11)	P8(0)
P5(9:11)	P1(0)	P6(9:11)	P7(0)
P8(12:17)	P1(0)	P6(12:13)	P7(14:17)
P9(12:17)	P2(0)	P5(0)	P8(12:17)

SEND the upper core region to the Right

Sending Processor:	Neighbours:		
P1(3:7)	P2(3:7)	P5(0)	P8(0)
P2(3:8)	P3(3:5)	P4(6:8)	P9(0)
P5(9:11)	P3(0)	P4(9:11)	P9(0)
P6(8:13)	P2(8:8)	P5(9:11)	P8(12:13)
P7(14:17)	P2(0)	P5(0)	P8(14:17)
P8(12:17)	P3(0)	P4(0)	P9(12:17)

Figure 3.8: Example demonstrating that the original communication message may not always start from CAP2_LOW, and end at CAP2_HIGH. The original communication set and the new DLB communication set are shown, along with the message range being sent and received by each processor with their neighbouring processors.

The starting index of the communicated data in the Staggered Dimension (FIRST) therefore needs to be passed in as an additional parameter of the communication call, from which the message end can be deduced (Figure 3.9). For example, FIRST=MAX(3,CAP2_LOW) for the Left halo communication

associated with Figure 3.8, which actually means that FIRST=3 on the first row of processors, and FIRST=CAP2_LOW on the second and third row of processors. If FIRST is not passed in through the call parameters then it would be difficult to establish the start (L) and the end (H) locations for the internal communications. Therefore the calculation of L and H in Figure 3.7 is replaced with those in Figure 3.9. Now for example, LOW=MAX(3,1) when updating the halo region on Processor 2 with the data stored on Processor 1, and HIGH=MIN(17,18) when updating the halo region on Processor 9 with the data from Processor 8. Note that the communication message length is already passed in through an existing parameter of the communication call (either NITEMS or NSTRIDE).

```
C       Obtain the start and end of this communication message for this processor
        L=FIRST
        H=FIRST+COMMUNICATION_MESSAGE_LENGTH-1
```

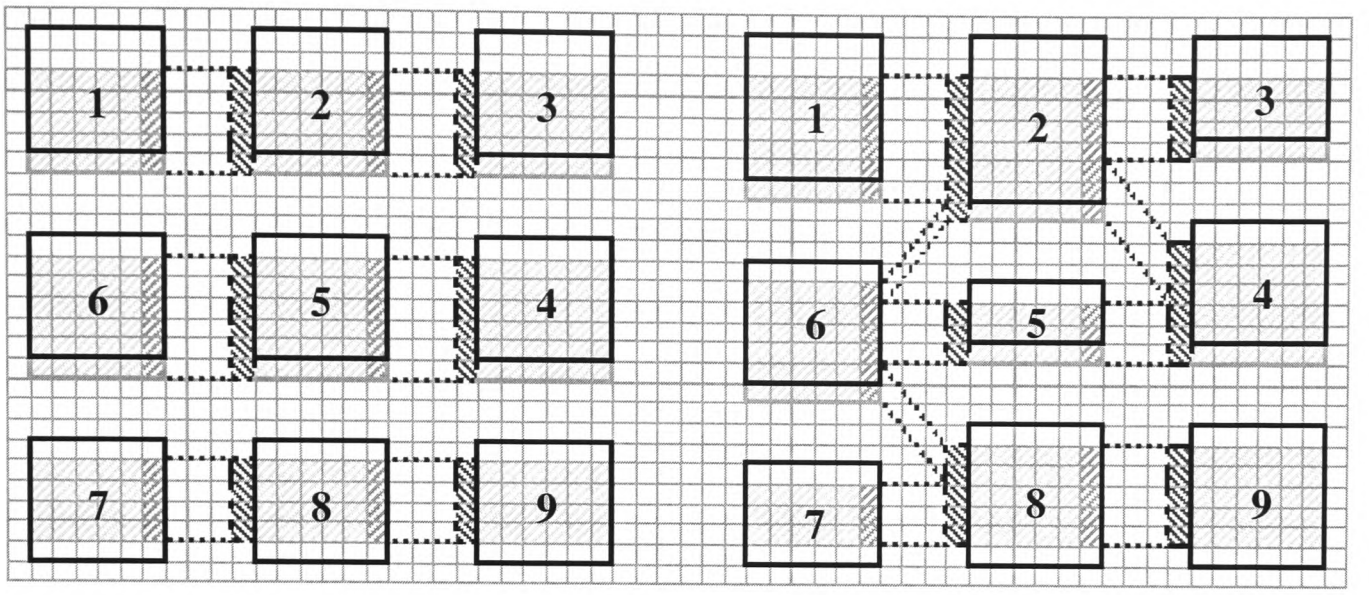
Figure 3.9: The communication start and end locations for the communicating processor, where FIRST is the starting index of the communicated data in the Staggered Dimension.

3.3.1.2 Communication Offsets

There are instances when a communication extends beyond the processor partition range limits, such as in the example demonstrated in Figure 3.10. The first row of processors will assign data between 3 and CAP2_HIGH+1, the middle row of processors will assign data between CAP2_LOW+1 and CAP2_HIGH+1, and the third row of processors will assign data between CAP2_LOW+1 and NJ-1. Some of the processors (those not in the last row) are assigning data in their halo region, which is then needed on neighbouring processors. For example, Processor 2 in Figure 3.10 needs to use T(6,3:9), where it needs to receive T(6,3:8) from Processor 1 and T(6,9) from Processor 6.

```
DO J=MAX(3,CAP2_LOW+1),
        MIN(NJ-1,CAP2_HIGH+1)
  DO I=CAP1_LOW,CAP1_HIGH
    T(I,J)=...
  END DO
END DO
```

```
DO J=MAX(3,CAP2_LOW+1),
        MIN(NJ-1,CAP2_HIGH+1)
  DO I=CAP1_LOW,CAP1_HIGH
    V(I,J)=T(I-1,J)
  END DO
END DO
```



RECEIVE into the lower halo region from the Left

Receiving Processor:	Neighbours:		
P2(3:9)	P1(3:8)	P6(9:9)	P7(0)
P3(3:6)	P2(3:6)	P5(0)	P8(0)
P4(7:12)	P2(7:9)	P5(10:12)	P8(0)
P5(10:12)	P1(0)	P6(10:12)	P7(0)
P8(13:17)	P1(0)	P6(13:14)	P7(15:17)
P9(13:17)	P2(0)	P5(0)	P8(13:17)

SEND the upper core region to the Right

Sending Processor:	Neighbours:		
P1(3:8)	P2(3:8)	P5(0)	P8(0)
P2(3:9)	P3(3:6)	P4(7:9)	P9(0)
P5(10:12)	P3(0)	P4(10:12)	P9(0)
P6(9:14)	P2(9:9)	P5(10:12)	P8(13:14)
P7(15:17)	P2(0)	P5(0)	P8(15:17)
P8(13:17)	P3(0)	P4(0)	P9(13:17)

Figure 3.10: Example demonstrating that the original communication message may be 'offset', such that a processor may assign data in their halo region, which is then needed by a neighbouring processor. The original communication set and the new DLB communication set are shown, along with the message range being sent and received by each processor with their neighbouring processors.

Using the existing algorithm (that now involves FIRST), Processor 2 would currently receive T(6,3:7) from Processor 1, and T(6,8:9) from Processor 6, where the staggered limits of Processors 1 and 6 have been compared against the message start and end.

The obvious problem in doing this is that the value of T(6,8) has not been assigned on Processor 6 (whose staggered limits actually include this range), but on Processor 1, meaning the usage (communication) of unassigned data. The assignment on Processor 1 in the Staggered Dimension ends at 8 (CAP2_HIGH+1) and the assignment on Processor 6 starts at 9 (CAP2_LOW+1), which means that the correct assigned data should be used on neighbouring processors otherwise an incorrect solution will be the result. The communication message is offset by +1 on both the lower and upper staggered limits in this example. Note that if the assignment had been made between MAX(3,CAP2_LOW-1) and MIN(NJ-1,CAP2_HIGH-2), then the offset on the lower staggered limit would be –1, and the offset on the upper staggered limit would be –2.

This suggests the need to modify the above algorithm even further, as shown in Figure 3.11, such that the communication message 'offsets' are involved in dissecting the original message. They are used to ensure that the operation of the DLB communication follows the exact operation of the original communication call, guaranteeing correctness of code. The lower 'message limit' in the Staggered Dimension (LOWLIM), and the upper 'message limit' in the Staggered Dimension (HIGHLIM), are therefore included in the DLB parameter list, as well as passing in FIRST. For example, in Figure 3.10 LOWLIM=CAP2_LOW+1, and HIGHLIM=CAP2_HIGH+1, where FIRST=MAX(3,CAP2_LOW+1). The values of LOWLIM and HIGHLIM are extracted from the loop limits involving the Staggered Dimension. The first parameters in the MAX and MIN are the original loop limits from the serial code which the boundary processors operate on, whereas the second set of parameters in the MAX and MIN are operated on by the intermediate processors. The first set of parameters can be ignored due to the usage of FIRST, which caters for the extreme values on the boundary processors. The message offsets (L_OFF and H_OFF) can then be calculated and applied to the staggered limits of neighbouring processors, which can then be compared against the message start and end (L and H).

On Processor 2 in Figure 3.10, for example, LOWLIM=2 and HIGHLIM=9, which means that L_OFF=(2-1)=1 and H_OFF=(9-8)=1. These offsets are then applied to the staggered limits of Processor 2's Left neighbouring

processors (1, 6, and 7), where the lower limit of Processor 1 is now NL=1+1=2 and its upper limit is NH=7+1=8. The lower limit of Processor 6 is NL=8+1=9 and its upper limit is NH=13+1=14, and similarly for Processor 7 whose lower limit is NL=13+1=15 and its upper limit is NH=18+1=19. These limits are each compared in turn with the message start (L=3) and the message end (H=9). A low-level communication between Processor 2 and Processor 1 will therefore be set up starting from LOW=MAX(3,2)=3 and ending at HIGH=MIN(9,8)=8. More importantly with this example, a low-level communication between Processor 2 and Processor 6 will be set up starting from LOW=MAX(3,9)=9 and ending at HIGH=MIN(9,14)=9. No communication occurs between Processor 2 and Processor 7 since LOW=MAX(3,15)=15 and HIGH=MIN(9,19)=9 (implies a negative communication length). Note that if LOWLIM and HIGHLIM are equal to the staggered limits then no offset is applied to the neighbouring processors.

```
C       Find 'offsets' from staggered limits
C       e.g.: lowlim=cap2_low+1 and highlim=cap2_high+1
C             => l_off=1 and h_off=1
        L_OFF=LOWLIM-CAP_DLB_PROCLIMITS(SD1,CAP_PROCNUM)
        H_OFF=HIGHLIM-CAP_DLB_PROCLIMITS(SD2,CAP_PROCNUM)
        ...

C       Obtain the start and end of the communication message for
C       the neighbouring processor, applying the offsets
        NL=CAP_DLB_PROCLIMITS(SD1,NEIGHBOUR)+L_OFF
        NH=CAP_DLB_PROCLIMITS(SD2,NEIGHBOUR)+H_OFF
```

Figure 3.11: Incorporating the lower and higher offsets into the algorithm.

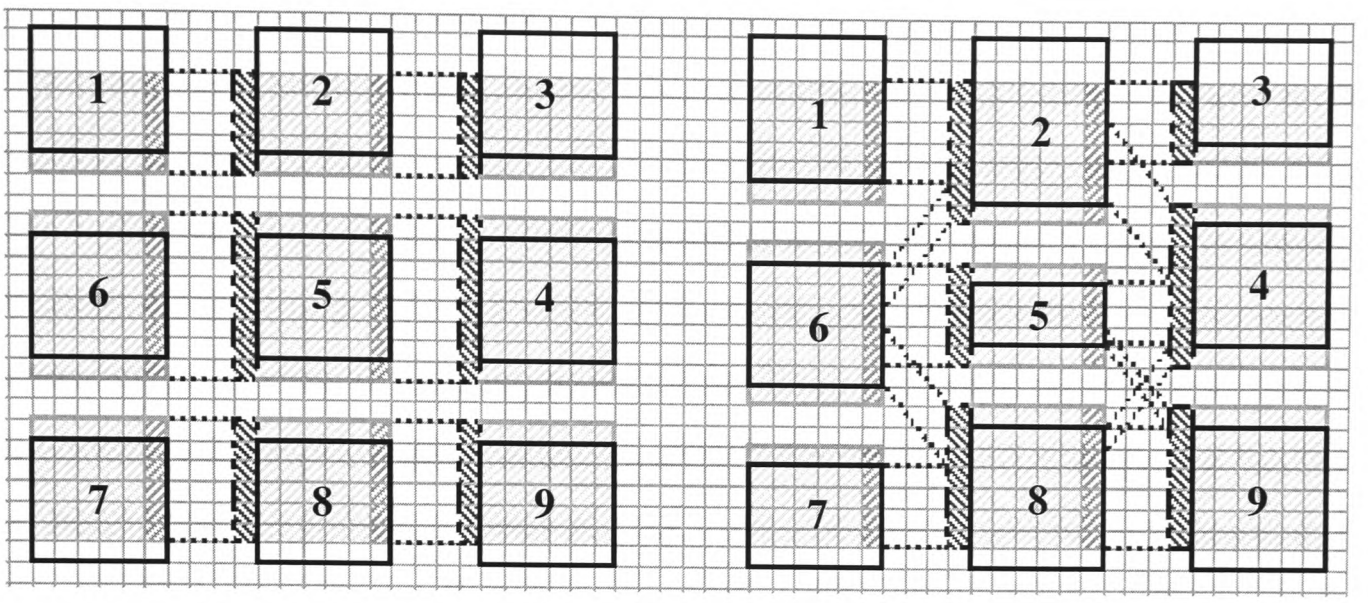
The example given in Figure 3.10 illustrates the situation in which data is always only assigned on one processor, however, the example shown in Figure 3.12 illustrates the situation in which data may be assigned on more than one processor. For example, with the staggered case in Figure 3.12, the value of T(6,8) is assigned on both Processor 1 and on Processor 6. With Figure 3.10 the requested data was received from the processor who made the assignment, which was easily identified since the data was only assigned on one processor. With Figure 3.12 the data is originally received and sent to an immediate neighbour, but using the above algorithm with the staggered case the same data could be communicated several times. For example, Processor 2 needs to receive T(6,3:9) from its Left (i.e. FIRST=3), where LOWLIM=CAP2_LOW-1 and

HIGHLIM=CAP2_HIGH+1, implying that L_OFF=-1 and H_OFF=1 using the algorithm in Figure 3.11. This means that Processor 2 will receive values of T between MAX(3,1-1)=3 and MIN(9,7+1)=8 from Processor 1, and will receive values of T between MAX(3,8-1)=7 and MIN(9,13+1)=9 from Processor 6. The problem with this is that cells T(6,7:8) are received twice by Processor 2.

In terms of sending data, Processor 1 needs to send T(6,3:8) to its Right (i.e. FIRST=3), where L_OFF=-1 and H_OFF=1. If using the algorithm in Figure 3.11, then Processor 2 will be sent values of T between MAX(3,1-1)=3 and MIN(8,8+1)=8 from Processor 1, and Processor 5 will be sent values of T between MAX(3,9-1)=8 and MIN(8,11+1)=8 from Processor 1. Similarly, Processor 6 will send values of T between MAX(7,1-1)=7 and MIN(14,8+1)=9 to Processor 2.

```
DO J=MAX(3,CAP2_LOW-1),
        MIN(NJ-1,CAP2_HIGH+1)
   DO I=CAP1_LOW,CAP1_HIGH
      T(I,J)=...
   END DO
END DO
```

```
DO J=MAX(3,CAP2_LOW-1),
        MIN(NJ-1,CAP2_HIGH+1)
   DO I=CAP1_LOW,CAP1_HIGH
      V(I,J)=T(I-1,J)
   END DO
END DO
```



RECEIVE into the lower halo region from the Left

Receiving Processor:	Neighbours:		
P2(3:9)	P1(3:7)	P6(8:9)	P7(0)
P3(3:6)	P2(3:6)	P5(0)	P8(0)
P4(5:12)	P2(5:8)	P5(9:11)	P8(12:12)
P5(8:12)	P1(0)	P6(8:12)	P7(0)
P8(11:17)	P1(0)	P6(11:13)	P7(14:17)
P9(11:17)	P2(0)	P5(11:11)	P8(12:17)

SEND the upper core region to the Right

Sending Processor:	Neighbours:		
P1(3:8)	P2(3:7)	P5(0)	P8(0)
P2(3:9)	P3(3:6)	P4(5:8)	P9(0)
P5(8:12)	P3(0)	P4(9:11)	P9(11:11)
P6(7:14)	P2(8:9)	P5(8:12)	P8(11:13)
P7(13:17)	P2(0)	P5(0)	P8(14:17)
P8(13:17)	P3(0)	P4(12:12)	P9(12:17)

Figure 3.12: Example demonstrating that the same data may be assigned on more than one processor. The original communication set and the new DLB communication set are shown, along with the message range being sent and received by each processor with their neighbouring processors.

When sending data using the current algorithm a comparison is made between the *sender* (communicating processor) and the *receiver* (neighbouring processor), L and Receiver_L2+L_OFF for example. The comparison of the *receiver* (communicating processor) and the *sender* (neighbouring processor) is

made when receiving data (L and Sender_L2+L_OFF for example). The current comparison can be seen in Figure 3.13, where L and H are the start and end of the communication message respectively (see Figure 3.9), and L2 and H2 represent the lower and higher staggered limits. As demonstrated by the example in Figure 3.12, this comparison allows data to be communicated more than once, which is why the modified algorithm uses a sender offset (SEND_OFF) to avoid this situation (Figure 3.13).

The value of SEND_OFF is based on the values of L_OFF and H_OFF, as summarised in Table 3.2. If L_OFF is positive and H_OFF is negative, or vice versa, or if both are 0, then SEND_OFF is set to 0, ensuring that the modified algorithm operates the same as the current algorithm where the staggered limits of the neighbour are compared with the start and end of the original communication message. The new start and end for the Receive communication will be affected since the neighbouring processor will only send data it owns (the offsets are ignored). For example, when Processor 2 in Figure 3.12 needs to receive T(6,3:9) from its Left, then SEND_OFF=0 since L_OFF=-1 and H_OFF=1, meaning it shall receive values of T between MAX(3,1+0)=3 and MIN(9,7+0)=7 from Processor 1, and values of T between MAX(3,8+0)=8 and MIN(9,13+0)=9 from Processor 6. Similarly, Processor 1 shall send values of T between MAX(3,1-1,1+0)=3 and MIN(8,8+1,7+0)=7 to Processor 2 on its Right, and Processor 6 shall send values of T between MAX(7,1-1,8+0)=8 and MIN(14,8+1,13+0)=9 to Processor 2.

If both L_OFF and H_OFF are positive then SEND_OFF is set to equal L_OFF, whereas if both are negative then SEND_OFF is set to equal H_OFF. This ensures that the correct data is sent (only sends data that it assigns) even if the offsets have different values, as demonstrated in Figure 3.14 for example where SEND_OFF would be set to 1 (as L_OFF=1 and H_OFF=2). Using the current algorithm, Processor 2 in Figure 3.14 would receive values of T between MAX(3,1+1)=3 and MIN(10,7+2)=9 from Processor 1, and values of T between MAX(3,8+1)=9 and MIN(10,13+2)=10 from Processor 6. Using the modified algorithm in Figure 3.13, Processor 2 now receives values of T between MAX(3,1+1)=3 and MIN(10, 7+1)=8 from Processor 1, and values of T between MAX(3,8+1)=9 and MIN(10, 13+1)=10 from Processor 6. More importantly, Processor 1 now sends values of T between MAX(3,1+1,1+1)=3 and

MIN(9,8+2,7+1)=8 to Processor 2, and Processor 6 now sends values of T between MAX(9,1+1,8+1)=9 and MIN(15,8+2,13+1)=10 to Processor 2. The reason why SEND_OFF is set to L_OFF for Figure 3.14 is clear when examining the Send communication on Processor 6 for instance, where it has to send data to Processor 8. Without the send offset, some data (T(6,15)) will be communicated twice, since the value of HIGH will be evaluated to MIN(15,17)=15 instead of MIN(15,18+2,13+1)=14.

Current algorithm:

Receive Communication:
LOW=MAX(L,Sender_L2+L_OFF)
HIGH=MIN(H,Sender_H2+H_OFF)

Send Communication:
LOW=MAX(L,Receiver_L2+L_OFF)
HIGH=MIN(H,Receiver_H2+H_OFF)

Modified algorithm:

Receive Communication:
LOW=MAX(L,Sender_L2+SEND_OFF)
HIGH=MIN(H,Sender_H2+SEND_OFF)

Send Communication:
LOW=MAX(L,Receiver_L2+L_OFF,Sender_L2+SEND_OFF)
HIGH=MIN(H,Receiver_H2+H_OFF,Sender_H2+SEND_OFF)

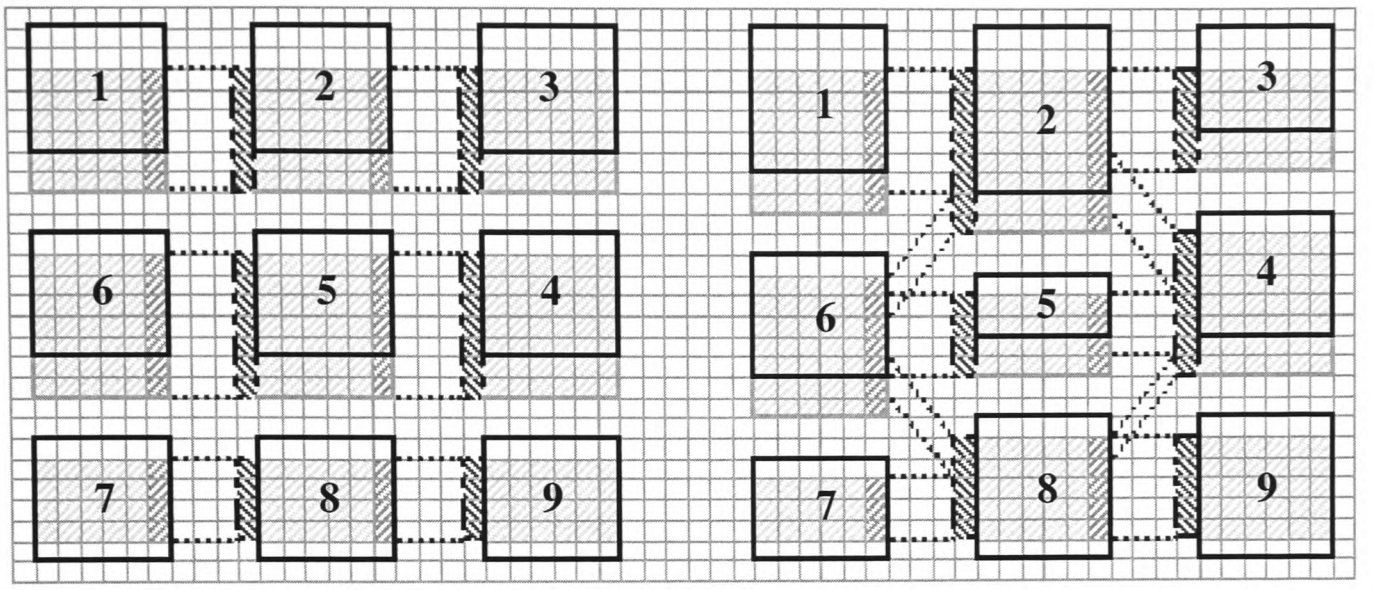
Figure 3.13: Current and modified algorithm that is used to determine the new communication message start (LOW) and end (HIGH), where L and H are the original communication start and end, and L2 and H2 are the staggered limits. A neighbouring processor is the sender in the Receive communication and is the receiver in a Send communication. L_OFF and H_OFF are used to determine the value of SEND_OFF that is used in the modified algorithm to avoid communicating the same data more than once.

L_OFF	H_OFF	SEND_OFF
0	0	0
+ve	-ve	0
-ve	+ve	0
+ve	+ve	L_OFF
-ve	-ve	H_OFF

Table 3.2: Evaluation of SEND_OFF based on the values of L_OFF and H_OFF.

```
DO J=MAX(3,CAP2_LOW+1),
        MIN(NJ-1,CAP2_HIGH+2)
  DO I=CAP1_LOW,CAP1_HIGH
    T(I,J)=...
  END DO
END DO
```

```
DO J=MAX(3,CAP2_LOW+1),
        MIN(NJ-1,CAP2_HIGH+2)
  DO I=CAP1_LOW,CAP1_HIGH
    V(I,J)=T(I-1,J)
  END DO
END DO
```



RECEIVE into the lower halo region from the Left

Receiving Processor:	Neighbours:		
P2(3:10)	P1(3:8)	P6(9:10)	P7(0)
P3(3:7)	P2(3:7)	P5(0)	P8(0)
P4(7:13)	P2(7:9)	P5(10:12)	P8(13:13)
P5(10:14)	P1(0)	P6(10:13)	P7(0)
P8(13:17)	P1(0)	P6(13:14)	P7(15:17)
P9(13:17)	P2(0)	P5(0)	P8(13:17)

SEND the upper core region to the Right

Sending Processor:	Neighbours:		
P1(3:9)	P2(3:8)	P5(0)	P8(0)
P2(3:10)	P3(3:7)	P4(7:9)	P9(0)
P5(10:13)	P3(0)	P4(10:12)	P9(0)
P6(9:15)	P2(9:10)	P5(10:13)	P8(13:14)
P7(15:17)	P2(0)	P5(0)	P8(15:17)
P8(13:17)	P3(0)	P4(13:13)	P9(13:17)

Figure 3.14: Example in which data is assigned on more than one processor, where L_OFF and H_OFF have different values but the same sign.

3.3.1.3 New Internal Starting Address

The original communication is sent from, or received into, a particular address in memory, but having split this communication into several new messages each will need to start from a unique position. For example, in Figure 3.10 Processor 6 will need to send T(6,9:14) to its Right, where this communication message needs to be internally dissected into three separate messages with Processor 2, Processor 5, and Processor 8. Therefore the new message will either start from the same location as the original message (which is 1 inside the utility routine), or will be offset by the difference between the original (FIRST) and the new (LOW) starting address, as demonstrated in Figure 3.15. The utility needs to operate in bytes to be applicable to several different data types, hence the communicated data type (ITYPE) is converted using CAP_TYPELENS. Since the dissected (staggered) index is not always contiguous in memory then it is necessary to stride over previous contiguous dimensions in order to reach the subsequent location in memory. Therefore the stride of the communicated data in the Staggered Dimension (STAG_STRIDE) needs to be passed into the utility through the parameter list, as it is not always known. For example, in Figure 3.10 if the stride of the second dimension (Staggered Dimension) equals 18, then the new starting address between Processor 6 and Processor 2 will be NEW_STARTING_ADD=1+(9-9)*18=1, where FIRST=9 and LOW=9. In this case the starting address is actually the same as the starting address of the original communication message. However, LOW=10 when Processor 6 needs to send data to Processor 5, which means that NEW_STARTING_ADD=1+(10-9)*18=19, and similarly NEW_STARTING_ADD=1+(13-9)*18=72 when sending to Processor 8.

```
NEW_STARTING_ADD=1+CAP_TYPELENS(ITYPE)*((LOW-FIRST)
                                        *ABS(STAG_STRIDE))
```

Figure 3.15: The new generic starting address, calculated in Bytes, is offset from the original starting address by a number of strides in the Staggered Dimension.

3.3.2 Splitting Buffered And Unbuffered Communications

The above has given a general overview of what occurs when communicating over non-coincidental limits, but in reality the dissection of the communication message length is dependent upon the type of communication, which are either buffered or unbuffered (Section A.3.3). Most CAPTools communication calls are based upon, or are a variation of, these two types of calls. We shall concentrate on these buffered and unbuffered communications and demonstrate that the additional parameters (FIRST, LOWLIM, HIGHLIM, and STAG_STRIDE) are sufficient to cope with both. It shall also become apparent that STAG_STRIDE has more than one use in these DLB communication utilities, since the utilities operate in 1D (where an index can be identified by its stride), minimising the need for any extra parameters.

Buffered communications are used to Send/Receive data that is contiguous in memory, and unbuffered are used to communicate disjointed continuous sections of data. The stride of the communicated data in the Staggered Dimension (STAG_STRIDE) is used to determine what operation is performed internally, dissecting either NITEMS or NSTRIDE of data.

With unbuffered communications, if the STAG_STRIDE is smaller than NITEMS (length of continuous items), then the communicated data will be affected by the staggered limits, resulting in the dissection of NITEMS itself, otherwise all of the continuous data should be communicated to a single neighbour. For example, consider the example shown in Figure 3.16a, where the continuous section needs to be dissected when communicated in the Up/Down direction, since the staggered stride (STAG_STRIDE=1) is smaller than the continuous length (NITEMS=20). In Figure 3.16b the staggered stride (STAG_STRIDE=30) is larger than the continuous length (NITEMS=20), which means that the whole message is communicated to the single processor, who in this instance owns that particular row of cells. This means that if a different row of data were communicated, then that row would be communicated with the neighbouring processor who also owned that row, since any portion of a row is owned by just one processor.

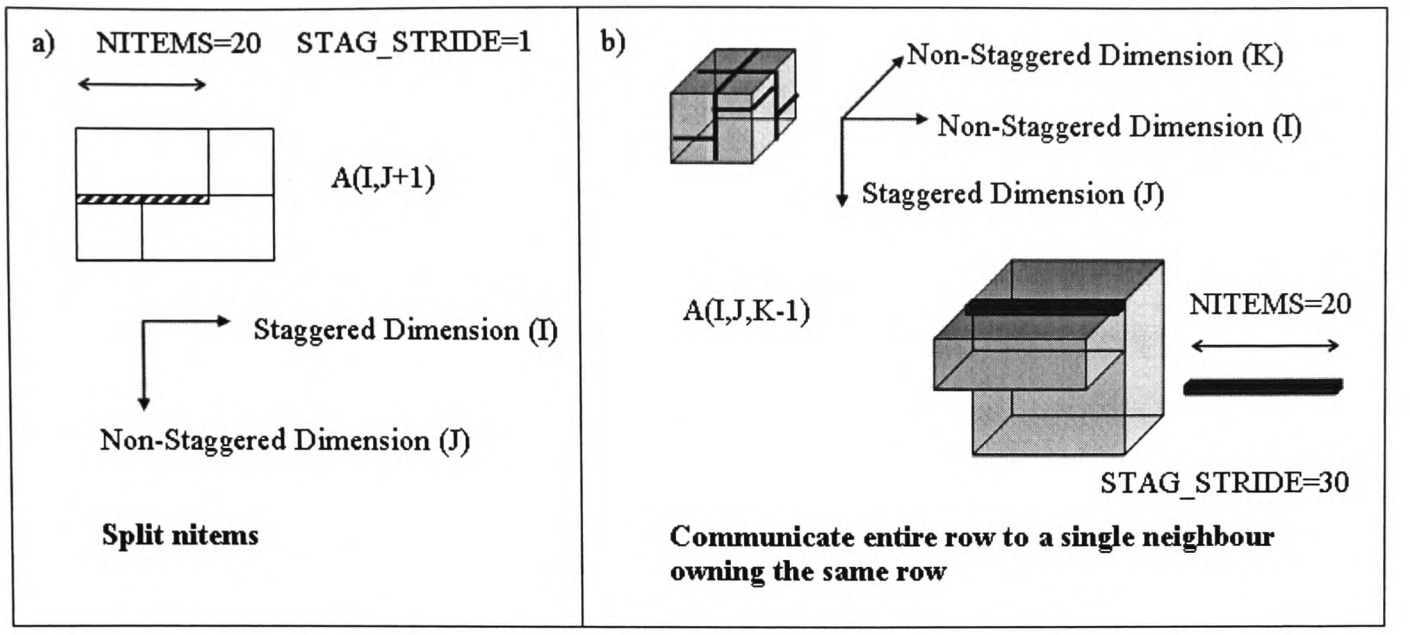


Figure 3.16: Unbuffered DLB communications in which a) the continuous message is dissected amongst neighbouring processors (STAG_STRIDE<NITEMS); and b) the continuous message is communicated with a single neighbour (STAG_STRIDE≥NITEMS). In both cases the length of the first dimension is 30.

With buffered communications, the comparison is made with the communication STRIDE (the length between successive blocks of continuous data), where NITEMS is dissected if the STAG_STRIDE is less than the communication STRIDE. If the STAG_STRIDE is the same as the communication STRIDE then this implies that NSTRIDE is dissected, since this essentially represents the communication length in the Staggered Dimension itself. Lastly, if the STAG_STRIDE is larger than the communication STRIDE then all of the buffered data is communicated to a single intersecting neighbour. The STAG_STRIDE is different for each communicated variable, which is another reason why it is necessary to include STAG_STRIDE as an additional parameter.

Figure 3.17 illustrates these three cases, where STAG_STRIDE equals 1 in case a), 30 in case b), and 600 in case c). The continuous data items in each of the two rows in Figure 3.17a need to be dissected in the same way as in Figure 3.16a, where the new buffered DLB communications will involve a share of the continuous items (retaining the STRIDE and NSTRIDE of the original communication). Therefore the first half of both rows of data will be received into Processor 1 from Processor 4, and the second half of both rows will be received from Processor 3. The number of rows (NSTRIDE) in Figure 3.17b need to be dissected amongst neighbouring processors, where the new buffered DLB communication will involve a share of the number of strides (retaining NITEMS

and the STRIDE of the original communication). Therefore the first few rows, of width=2, will be received by Processor 4 from Processor 2, where the remaining rows will be received from Processor 3. The entire plane, of width=15 and height=2, in Figure 3.17c will need to be communicated with the processor whose staggered limits contain that particular plane of data, as the buffered message is not dissected by any staggered limits. In this instance, Processor 2 contains this portion of the first plane (the plane containing the buffered data), and so Processor 3 will receive all of the buffered data from Processor 2. Note that in Figure 3.17c NITEMS, the STRIDE, and NSTRIDE, all remain the same as in the original buffered communication.

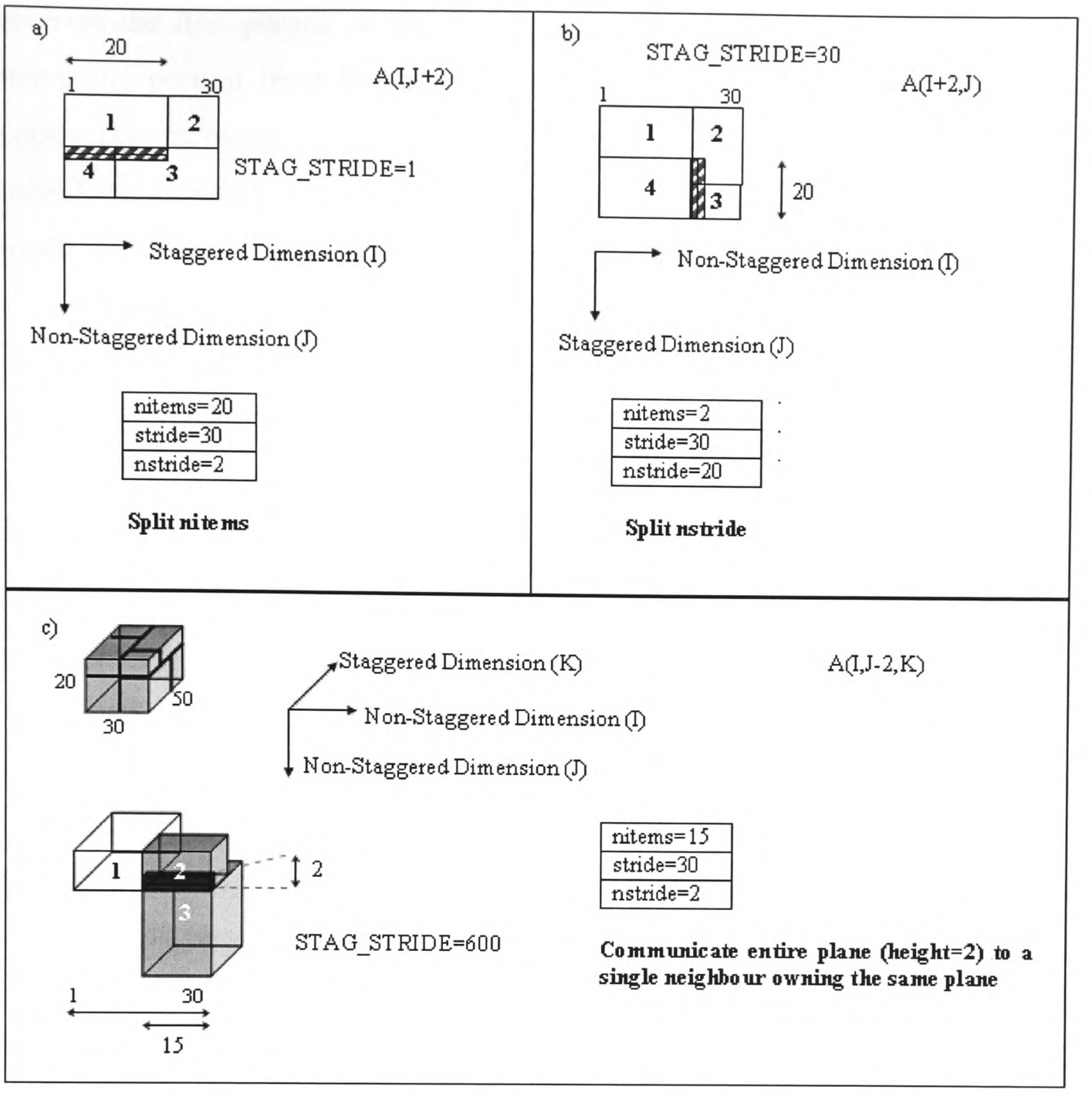


Figure 3.17: Buffered DLB communications in which a) the continuous message is dissected amongst neighbouring processors (STAG_STRIDE<STRIDE); b) the number of strides between successive continuous blocks of data is dissected amongst neighbouring processors (STAG_STRIDE=STRIDE); and c) the buffered message is communicated with a single neighbour (STAG_STRIDE>STRIDE). In each case the length of the first dimension is 30, and the length of the second dimension is 20.

As stated earlier, the communication library must work using one-dimensional data in order to be generic, where a particular index can be specified by its stride (for example, the Staggered Dimension index can be given as the input parameter STAG_STRIDE). Figure 3.18 shows the memory layout for the buffered communications shown in Figure 3.17, demonstrating how the original communication message is dissected inside a DLB communication call. The STAG_STRIDE=1 in Figure 3.18 is smaller than the buffered stride (STRIDE=30), which means that two rows of 20 items of contiguous data (NITEMS) are received from different neighbouring processors. Processor 1

receives the first portion of the 20 continuous items from Processor 4, and the remaining portion from Processor 3, where this pattern is duplicated in every contiguous memory block for each stride through memory. Note that with unbuffered communications the same dissection would occur, but only on a single block.

The second line of memory shown in Figure 3.18b relates to the case when the STAG_STRIDE is the same as the buffering stride. In this case, all of the items in each contiguous block of memory are received from the same processor, but the different blocks are received from different processors. Processor 4 receives the first few blocks of 2 continuous items from Processor 2, after which the remaining blocks of 2 are received from Processor 3. Finally, Figure 3.18c demonstrates that Processor 3 receives the entire communication from Processor 2, since the staggered stride (600) is larger than the buffered stride (30).

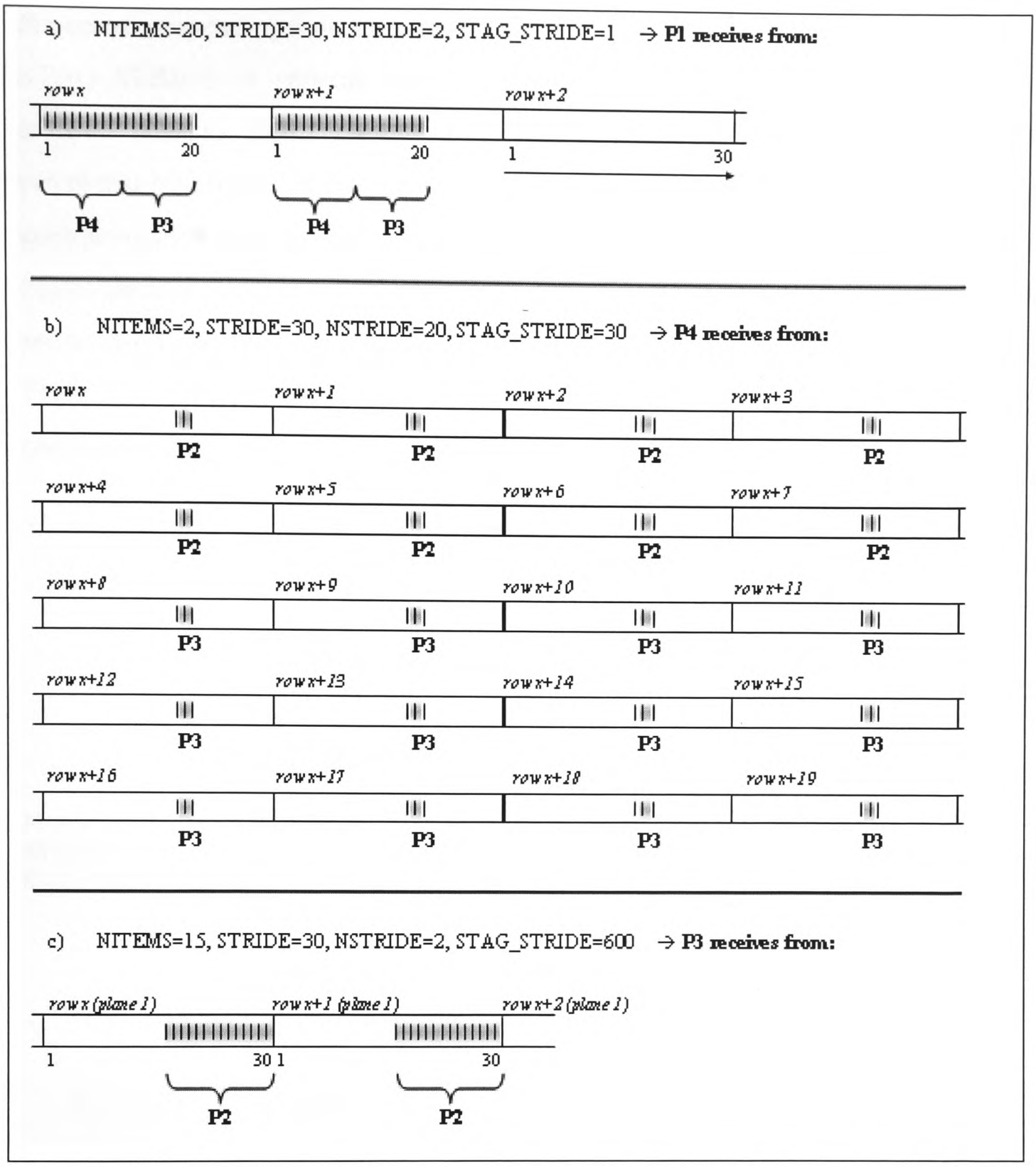


Figure 3.18: One-dimensional memory map of the buffered communications shown in Figure 3.17 with a) the staggered stride less than the buffering stride; b) the staggered stride equal to the buffering stride; and c) the staggered stride greater than the buffering stride. The neighbouring processors involved in the communication are shown below each memory line.

Note that it may be possible for a buffered communication to stride backwards through memory (when the STRIDE is negative), in which case the start and end of the communication message (shown in Figure 3.9) need to be swapped around, as demonstrated in Figure 3.19. The starting index, passed in as FIRST, is in fact the end index of the communication message, which should then be internally compared against the upper staggered processor partition range limit rather than the comparing the upper staggered processor partition range limits against the actual message start index. Either NITEMS or NSTRIDE may be

dissected depending on the type of communication call and on the STAG_STRIDE. A general term (COMMUNICATION_MESSAGE_LENGTH) is used here to depict either the newly calculated NITEMS, or the newly calculated NSTRIDE. If the communication stride (STRIDE) is negative, then the communication starting address (FIRST) will involve the last index of the communicated variable, in which case the internal communication message end is set to this value (H). A summary of the various values of L and H are given in Table 3.3, where it is evident that the communication stride for the unbuffered communication is always 1.

```
C     Obtain the start and end of this communication message for this
C     processor when the STRIDE is –ve
      IF( STRIDE.GT.0 )THEN
        L=FIRST
        H=FIRST+COMMUNICATION_MESSAGE_LENGTH-1
      ELSE IF(STRIDE.LT.0 )THEN
        H=FIRST
        L=FIRST-COMMUNICATION_MESSAGE_LENGTH+1
      END IF
```

Figure 3.19: The communication start and end for the communicating processor when the STRIDE is negative, where FIRST is the starting index of the communicated data in the Staggered Dimension.

Unbuffered	L	H
STRIDE=1 (always)	FIRST	FIRST+NITEMS+1

Buffered	STRIDE	L	H
STAG_STRIDE < STRIDE	+ve	FIRST	FIRST+(NITEMS-1)/STAG_STRIDE
	-ve	FIRST-(NITEMS-1)/STAG_STRIDE	FIRST
STAG_STRIDE = STRIDE	+ve	FIRST	FIRST+NSTRIDE-1
	-ve	FIRST-NSTRIDE+1	FIRST
STAG_STRIDE > STRIDE	+ve	FIRST	FIRST
	-ve	FIRST	FIRST

Table 3.3: Summary of the various values of L (new message start) and H (new message end), depending on the sign of communication stride (STRIDE).

Figure 3.20 gives a general overview of the operations performed in the new DLB communications, along with the appropriate low-level Send communications, which can be applied to many other CAPTools generated communications since they are usually a variation of either buffered or unbuffered

communications. Therefore, the underlying operations performed by DLB communications can be encapsulated by this overview and the modified algorithm demonstrated in Section 3.3.1. Note that the communication STRIDE used in the low-level communication for a buffered communication is always positive when STAG_STRIDE ≤ STRIDE, as the message start and end have already been swapped around if the STRIDE was negative (Figure 3.19). The communication message is not dissected when STAG_STRIDE > STRIDE, which means that it is still possible to have a negative stride.

Unbuffered communication calls:-

```
IF( STAG_STRIDE < NITEMS )THEN
   Communicate small sections of the continuous length to various neighbours
   by dissecting NITEMS
   CAP_LOW_SEND(A(NEW_STARTING_ADD),NEW_LENGTH,
                        ITYPE,NEIGHBOUR)
ELSE
   Communicate a single unit in the Staggered Dimension with one neighbour
   CAP_LOW_SEND(A,NITEMS,ITYPE,NEIGHBOUR)
END IF
```

Buffered communication calls:-

```
IF( STAG_STRIDE < STRIDE )THEN
   Communicate small sections of the continuous length to various neighbours
   by dissecting NITEMS
   CAP_LOW_BSEND(A(NEW_STARTING_ADD),NEW_LENGTH,
                        ABS(STRIDE),NSTRIDE,ITYPE,NEIGHBOUR)
ELSE IF( STAG_STRIDE = STRIDE )THEN
   Communicate continuous sections to various neighbours
   by dissecting NSTRIDE
   CAP_LOW_BSEND(A(NEW_STARTING_ADD),NITEMS,ABS(STRIDE),
                        NEW_LENGTH,ITYPE,NEIGHBOUR)
ELSE IF( STAG_STRIDE > STRIDE )THEN
   Communicate a single unit in the Staggered Dimension with one neighbour
   CAP_LOW_BSEND(A,NITEMS,STRIDE,NSTRIDE,NEIGHBOUR)
END IF
```

Figure 3.20: Overview of dissection of communication messages for both unbuffered and buffered communications, where an example of the appropriate low-level Send communications is also given.

The stride of a particular array index can be found by calculating the product of the dimension size of previous indices, since the stride relates to an index dimension (Figure 3.21). This means that two strides of an array are related, so they are either the same, or one is a factor of the other. The communication

message length can be dissected exactly, since the STAG_STRIDE is a component of the message (it is not just a fraction of the message, but a factor).

$$STRIDE_X = \prod_{i=1}^{Index_X} n_{i-1} \qquad \text{where } n_0=1 \text{ and } n_i=\text{size of the } n^{th} \text{ dimension}$$

$$STRIDE_1 < STRIDE_2 \cdots \Rightarrow \cdots STRIDE_1 \times \prod_{i=N_1+1}^{N_2} n_{i-1} = STRIDE_2$$

$$STRIDE_1 > STRIDE_2 \cdots \Rightarrow \cdots STRIDE_2 \times \prod_{i=N_2+1}^{N_1} n_{i-1} = STRIDE_1$$

Figure 3.21: Equating the strides of different dimensions for an array variable.

3.3.3 The New DLB Communication Utilities

The DLB communication utilities appear similar in structure to existing communication call utilities, implying minimal changes will be made to the user's code. An example of both unbuffered and buffered communications along with their corresponding new DLB communication calls are given in Figure 3.22. The call name now signifies that the original communication message may now be split into several internal communications in a Non-Staggered Dimension, where only four additional parameters are needed, keeping the number of additional parameters to a minimum. All of the additional parameters relate only to the Staggered Dimension, where FIRST is either the starting index of an array in the Staggered Dimension, or it is the execution control mask value in the Staggered Dimension. STAG_STRIDE is either the stride of the Staggered Dimension, or it is set to 0 for 'special' DLB communications (Section 3.3.4). LOWLIM and HIGHLIM are the message boundaries, usually taking the values of the staggered processor partition range limits themselves. The extra parameters are added before ITYPE, as they relate to the message length. The user should still be able to understand the purpose of this call, but should also be able to easily distinguish it from the existing communication calls, in which a variable (A) of a certain type (ITYPE) is communicated in a specified direction (PID). NITEMS is the length of

a contiguous section of memory, and STRIDE and NSTRIDE are used in buffered communications where multiple sections of contiguous blocks of memory are communicated. Note that similar changes need to be made to other communication calls that are used within CAPTools.

```
CAP_SEND(A,NITEMS,ITYPE,PID)
CAP_DLB_SEND(A,NITEMS,FIRST,STAG_STRIDE,LOWLIM,HIGHLIM,
ITYPE,PID)

CAP_BSEND(A,NITEMS,STRIDE,NSTRIDE,ITYPE,PID)
CAP_DLB_BSEND(A,NITEMS,STRIDE,NSTRIDE,FIRST,STAG_STRIDE,
LOWLIM,HIGHLIM,ITYPE,PID)
```

Figure 3.22: Existing unbuffered and buffered communication calls alongside the new DLB communication calls, in which four extra parameters have been including.

The following example is used in conjunction with Figure 2.4 to demonstrate the use of the DLB communications described above, where a 3D array, U(Dim1,Dim2,Dim3), has been partitioned in the manner described in Table 3.4 in which dimension 2 is the Staggered Dimension. The original buffered communication call in which the halo region (cap3_high+1) is updated from the Right is shown in Figure 3.23, along with its replacement DLB call (as this call is in a Non-Staggered Dimension) that follows the format given in Figure 3.22.

Index	Partition Number (Pass)	Partition range limit	
		Low	High
1	3	cap3_low	cap3_high
2 **(SD)**	2	cap2_low	cap2_high
3	1	cap1_low	cap1_high

Index	Length	Stride	Direction
1	cap3_high – cap3_low + 1	1	Left/Right
2	cap2_high – cap2_low + 1	Dim1	Up/Down
3	cap1_high – cap1_low + 1	Dim1 x Dim2	Back/Forth

Table 3.4: Shows the partition information for the variable U, where the second index has been staggered (Staggered Dimension created on pass 2).

cap_breceive	(U(cap3_low,cap2_low,cap1_high+1), (cap3_high-cap3_low+1),dim1,(cap2_high-cap2_low+1), 2,cap_right)
cap_**dlb**_breceive	(U(cap3_low,cap2_low,cap1_high+1), (cap3_high-cap3_low+1),dim1,(cap2_high-cap2_low+1), **cap2_low,dim1,cap2_low,cap2_high**,2,cap_right)

Figure 3.23: The original and new DLB communication calls are given for updating the halo region shown in Figure 2.4 in Section 2.5.3.

In this example the halo region on Processor 6 is being updated from the Right, which originally meant receiving this data from just Processor 5 (which is its immediate Right neighbour), but with the DLB Staggered Limit Strategy it needs to receive this data from Processors 2, 5, and 8. Considering the case in which the partition range variables on each of the processors concerned are as given in Figure 3.24, then the three low-level communications (also shown), would be executed internally by the single DLB communication call. Note that $cap2\_low_5$ is the value of the CAP2_LOW on processor 5.

Processor 2 : $cap2\_high_2$ = 7
Processor 5 : $cap2\_low_5$ = 8 $cap2\_high_5$ = 10
Processor 6 : $cap2\_low_6$ = 7 $cap2\_high_6$ = 12
Processor 8 : $cap2\_low_8$ = 11

Receive U(cap3_low:cap3_high,$cap2\_low_6$:$cap2\_high_2$,cap1_high+1) -> from Processor ***2***
Receive U(cap3_low:cap3_high,$cap2\_low_5$:$cap2\_high_5$,cap1_high+1) -> from Processor ***5***
Receive U(cap3_low:cap3_high,$cap2\_low_8$:$cap2\_high_6$,cap1_high+1) -> from Processor ***8***

Figure 3.24: Shown are the staggered processor partition range limits for the processors involved in the DLB communication shown in Figure 3.23, where the internally executed low-level communications are shown.

3.3.4 'Special' DLB Communications

Some communications may only be executed within an execution control mask, where only those processors where the mask is true will need to communicate with each other. In Figure 3.25 for example, the value of V(CAP1_LOW-1,8) needs to be known on those processors owning row 8, where this would have originally involved immediately neighbouring processors with the non-DLB

communication, but will now involve whichever processor owns row 8 in the adjacent column of processors.

```
IF( 8.LE.CAP2_HIGH .AND. 8.GT.CAP2_LOW )THEN
   CALL CAP_DLB_EXCHANGE(V(CAP1_LOW-1,8),V(CAP1_HIGH,8)1,
                                 8,0,CAP2_LOW,CAP2_HIGH,2,CAP_LEFT)
END IF
```

Figure 3.25: Example demonstrating a 'special' DLB communication in which only those processors owning row 8 will be involved.

Using the current algorithm for the DLB communication, although only those processors owning row 8 will actually perform the communication due to the execution control mask, each processor will still try to communicate their overlapping region in the Staggered Dimension. In this instance, the communication only needs to involve row 8, and so this value needs to be passed into the DLB communication utility to maintain the original operation of the original communication.

The owner of the assigned data can be identified using the execution control mask value (8 in this instance), passed in as FIRST, which is used to determine which processors need to be involved in the communication. The staggered processor partition range limits of the communicating processor are first compared against FIRST to establish whether they are involved in the communication, as the communication may not always be contained within an execution control mask. After ascertaining the neighbouring processors in the communication direction (PID), FIRST is then compared against the staggered processor partition range limits of these neighbours, where a low-level communication call is set up between the identified processors. As no changes are required to the original communication message, the internal starting address (NEW_STARTING_ADD in Figure 3.15) will not need to be offset, meaning that STAG_STRIDE should be set to zero in order for the new communication message to start from the same location in memory (1). Therefore it is possible to handle this type of 'special' situation without the need to introduce any more parameters to the call list, keeping code changes to a minimum.

The algorithm for the DLB communication utility can be modified even further to cater for this 'special' type of situation which is signified by STAG_STRIDE=0 (Figure 3.26). Only one parameter is sufficient to identify

'special' DLB communications, making STAG_STRIDE the most likely candidate, meaning LOWLIM and HIGHLIM are redundant in this type of DLB communication.

```
      IF( STAG_STRIDE.EQ.0 )THEN
C       This is a masked communication
C       Communicate only if you own the masked value
        IF( FIRST.GE.CAP_DLB_PROCLIMITS(SD1,CAP_PROCNUM)
     +  .AND.FIRST.LE.CAP_DLB_PROCLIMITS(SD2,CAP_PROCNUM) )
     +  THEN
C         Find neighbour who also owns the masked value
          DO I=1,NUMBER OF NEIGHBOURS
C           Obtain neighbour i in the given direction
            NEIGHBOUR = ALLNEIGHBOURS(I,PID)
            IF( NEIGHBOUR.NE.0 )THEN
              IF( FIRST.GE.CAP_DLB_PROCLIMITS(SD1, NEIGHBOUR)
     +          .AND.FIRST.LE.CAP_DLB_PROCLIMITS(SD2,NEIGHBOUR)
     +          )THEN
C               Have found neighbour who owns data
C               Communicate message using a low-level communication
C               call
C               No need to process any more neighbours, as
C               communication is completed
                GOTO 10
              END IF
            ELSE
C             No neighbours in this direction
              GOTO 10
            END IF
          END DO
10        CONTINUE
        END IF
      ELSE
C       This is not a masked communication – perform normal DLB
C       communication
      END IF
```

Figure 3.26: 'Special' DLB communications that do not dissect the communication message but determine who to communicate with based on the execution control mask of the assigned data (passed in as FIRST).

3.3.5 Testing The DLB Communication Utilities

The DLB communications were tested on a number of CAPTools generated codes by manually altering the necessary communications throughout the code, such that they were now DLB communication calls. The functionality of the DLB communications were tested by manually changing the processor partition range limits in the code (either by hard coding the limits into the code, or by using a

debugger). The processor limits were initially staggered for this purpose, ensuring that there was no need to migrate any data (which can be tested separately). The DLB communications were believed to be correct if the same data was communicated as in the original parallel code generated by CAPTools.

If these newly developed DLB communication utilities were not available for use, then the user's code would simply become cluttered with 'DLB' code. The above algorithm (DLB code) would need to be inserted in place of each of the existing corresponding communications, for every communicated variable. For example, a single unbuffered communication involving the variable T would need to be replaced by a variation on the algorithm for the DLB unbuffered communication (also involving T), where a similar block of code would be introduced for the other communications in the user's code. DLB variables would have to be declared in the user's code, making the original application code less visible to the user, hindering further maintenance and optimisation.

The example shown in Figure 3.27 is an extract of sample code that would need to be generated if the DLB communications were not used, where several statements are now needed to update just one halo region on Processor 5 from the Right. Processor 5 originally receives its upper halo region from Processor 4 (when using a 3x3 processor topology), but may now receive this from Processor 3, 4, and 9, when staggered limits are implemented. In the given example the processor topology is fixed, such that the code would have to be modified if a different topology were used. Note that usually the halo region is updated on all of the processors, and so several statements would be needed for each processor, where the number could increase or decrease depending on the overlapping neighbouring processors.

Given partition limits: L1=CAP1_LOW H1=CAP1_HIGH
L2=CAP2_LOW H2=CAP2_HIGH

Non-DLB communications:

```
CALL CAP_BRECEIVE5  (A(H1+1,L2),1,NI,H2-L2+1,2,CAP_RIGHT)
  CALL CAP_BSEND4   (A(L1,L2)1,NI,H2-L2+1,2,CAP_LEFT)
```

Transformation into DLB communications:

```
CALL CAP_BRECEIVE5  (A(H1+1,MAX(L25,L23)),1,NI,
                    MIN(H25,H23)-MAX(L25,L23)+1,2,CAP_RIGHT)
  CALL CAP_BSEND3   (A(L1,MAX(L25,L23)),1,NI,
                    MIN(H25,H23)-MAX(L25,L23)+1,2,CAP_LEFT)

CALL CAP_BRECEIVE5  (A(H1+1,MAX(L25,L24)),1,NI,
                    MIN(H25,H24)-MAX(L25,L24)+1,2,CAP_RIGHT)
  CALL CAP_BSEND4   (A(L1,MAX(L25,L24)),1,NI,
                    MIN(H25,H24)-MAX(L25,L24)+1,2,CAP_LEFT)

CALL CAP_BRECEIVE5  (A(H1+1,MAX(L25,L29)),1,NI,
                    MIN(H25,H29)-MAX(L25,L29)+1,2,CAP_RIGHT)
  CALL CAP_BSEND9   (A(L1,MAX(L25,L29)),1,NI,
                    MIN(H25,H29)-MAX(L25,L29)+1,2,CAP_LEFT)
```

Figure 3.27: Example code showing the original communication between Processor 5 and Processor 4, and the new code needed when staggered limits are implemented, where Processor 5 may have to communicate with Processors 3, 4, and 9, when using a 3x3 processor topology.

The DLB communication utilities allow the user to implement the DLB Staggered Limit Strategy within their code much easier than if the utilities were not available, since only minor changes are needed to the existing CAPTools generated parallel code, rather than major rewrites.

Having successfully devised and tested the above DLB communication utilities (with negligible overheads over the non-DLB equivalents), it is now possible to concentrate on those utilities that actually enforce DLB within a code, such as deciding when to redistribute the workload, how much to redistribute, and physically redistributing the workload between the processors. The DLB communications enable the DLB Staggered Limit Strategy to be implemented, as they allow processors to communicate over non-coincidental processor partition range limits, whereas the following utilities focus on redistribution itself.

3.4 Determine When To Redistribute

As mentioned in Section 2.7.2, the issue of when to balance the workload can affect the performance of the user's parallel code. If the workload is not redistributed frequently enough then the load imbalance and idle time can become significant, whereas redistributing the workload too often can lead to the redistribution time becoming significant. A compromise is needed where the workload is only redistributed if the cost of load imbalance outweighs the cost of redistributing the workload, i.e. redistribute the workload if $Cost_{Load\ Imbalance} > Cost_{Redistribution}$. However, the workload should not be redistributed simply because the redistribution time is low, and neither should the load be redistributed just because the idle time is high, which is why other factors need to be considered.

A decision needs to be made regarding whether or not the load should be balanced at the current iteration of some imbalanced loop, given the current level of load imbalance. The model of computation, discussed in Section 2.7.2.2, can be used to determine how frequently to redistribute the load. Although it is possible to estimate the level of load imbalance in subsequent iterations using the model of computation, the actual level of load imbalance in these iterations may change dramatically due to the physical characteristics of the code. As with the case of physical imbalance, discussed in Section 1.11.2.2, a particular iteration may be computationally intensive compared to the previous iteration of the same loop. Similarly, it is unlikely that all of the load imbalance will ever be removed after load redistribution, as the granularity of the problem influences how much load can be moved onto another processor, as it is not possible to move single cells (an entire row, say, will be moved, see Section 2.2).

The utility that decides when to perform the next redistribution is called CAP_DLB_DECIDE, which uses the processor timings in evaluating the model of computation. The processor timings are evaluated within the utility, rather than being placed in the user's code, to minimise the changes to the user's code, and additionally because the utility can then be used for a wide range of application codes. The maximum processor computation time is used in this model, since it is the timing of the slowest/heaviest processor that affects the performance of the user's code and not the average or minimum timing. However, the average

processor timing is used in calculating the rate of load imbalance (B), as illustrated in Figure 3.28, which is used in determining when to redistribute the load (see Figure 2.9). The rate of load imbalance can also be referred to as the proportion of idle time, which is equivalent to the proportion of idle time divided by the time since the last redistribution. If the maximum processor timing were the same as the average processor timing then this would imply that there is no load imbalance present in the system of processors.

B=(MAX_TIME-AVE_TIME)/TIME_SINCE_LAST_REBALANCE

Figure 3.28: Calculating the rate of load imbalance (B).

The utility CAP_DLB_DECIDE stops timing the imbalanced loop (with a call to CAP_DLB_STOP_TIMER) and decides whether or not to redistribute the load (see Section 4.4). The time spent computing, for each processor, in the timed section of code is calculated using the difference between the execution time and the time spent communicating (which includes idle time). The maximum and average of the processor computation times are obtained, after which the rate of load imbalance is evaluated, having already incremented the number of iterations. The number of iterations is evaluated internally, because the iteration counter variable may differ from code to code, and so it is more generic to evaluate this internally instead of having to decide which application code variable is the loop counter. If the maximum and average computation times were the same then the problem would be perfectly balanced, otherwise the aim is to reduce this maximum computation time by redistributing the load.

The algorithm determines in how many iterations, after the last redistribution, the load should be balanced. CAP_DLB_N_REBAL (n in Figure 2.9) returns the solution to this, which, when added onto the iteration number of the previous redistribution, gives the estimated iteration number at which the load should next be redistributed. The load should be redistributed only if the estimated redistribution iteration number is less than, or equal to, the current iteration number, as this indicates that a redistribution may prove profitable given the current level of load imbalance. Additionally, to avoid the load being redistributed unnecessarily due to a temporary surge in processor usage (interference by other users/jobs), the ratio of the maximum and average processor timings is used to

prevent redistribution when less than 1.16 (see Section 4.9.1). This constraint is not necessary when assuming physical imbalance on a homogeneous system such as the T3E for instance because such noise would not exist due to exclusive usage of the processors involved.

CAP_DLB_START_TIMER, CAP_DLB_START_REBAL and CAP_DLB_STOP_REBAL are all utilities that are used to obtain the timings needed to make the above decision regarding when to balance the load. The first utility is used to start the timers of the imbalanced iteration loop, which is executed before any statements of the load imbalanced code. CAP_WALLCLOCK_SECOND returns the number of wallclock seconds (real time) since the first call, and CAP_COMM_SECOND returns the number of seconds spent communicating since the first call. The second and third utilities time the load redistribution process and are only executed if it has been decided that the load should be redistributed (which was determined in CAP_DLB_DECIDE). CAP_DLB_REBAL_TIME (R in Figure 2.9) can now be used to determine when to balance the load in a later iteration of the DLB Loop. If the cost of redistributing the load is initially set as free (redistribution time set to zero) then this will encourage the load to be redistributed in an early iteration.

3.5 Calculate The New Processor Partition Range Limits

The workload on each processor can be defined in terms of the processor partition range limits, which if changed will alter the processor workload. Using the assumptions and constraints discussed in Section 2.7.4, the new processor partition range limits can be obtained by first calculating the new load on each processor, and then actually evaluating the new limits (Section 3.5.2).

The processor calculating the new processor partition range limits is arbitrarily chosen, for example every processor or just one processor could perform the calculation. In the current implementation only Processor 1 performs these calculations as this was the easiest to implement.

Each partitioned dimension is processed separately, since the processor partition range limits of each partitioned dimension are independent from one

another. It makes sense to balance the Non-Staggered Dimensions before the Staggered Dimension since the balance obtained in the later dimension 'fine tunes' the balance obtained in the previous dimensions. Therefore the Non-Staggered Dimensions are processed first, followed by the Staggered Dimension containing non-coincidental processor partition range limits.

The processor partition range limits in a Non-Staggered Dimension are the same for a group of processors, as global limits are used, and need to remain so. In Figure 3.5 for instance, the Left/Right limits are the same for Processors 2, 5 and 8 that are in the same column of processors, and similarly the Left/Right limits are the same for the processors in each of the other columns of processors. This means that each row of processors cannot be balanced separately but must be balanced collectively as a column of processors that will share the same Left/Right limits. The new Left/Right limits therefore need to be calculated just once for the three rows of processors, ensuring that the processors in each column of processors share the same limits.

Each column of processors is then processed separately (independently from one another), as the Up/Down limits can differ for every processor within a column of processors. Referring again to the example shown in Figure 3.5, the new Up/Down limits are calculated for the processors in the first column of processors (Processors 1, 6 and 7), then the second (Processors 2, 5 and 8), and so on. Processors no longer need to be grouped together, implying that there is a subtle difference compared to calculating the new limits for a Non-Staggered Dimension.

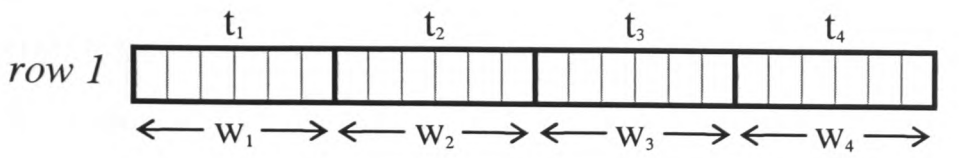
3.5.1 Calculating The New Workload On Each Processor

It has already been decided that each partitioned dimension shall be processed separately, and so the next stage is to decide how much to move. A particular iteration suspected of containing load imbalance within the application code is timed, where the processor timings are used to calculate the new distribution of cells. The computation time for each processor is returned, from which a weight (time per cell) can be obtained. For example, consider the single row of cells

shown in Figure 3.29a, where the width of cells on each processor are shown along with the time to process those cells (w_p and t_p respectively). If each processor had thousands of cells then it would be very costly to actually time every cell on each processor, which is why it is desirable to simply time the entire width of cells just once and use this in the calculation to find the new Left/Right limits. The estimate for the time per cell on a processor is given as the processor timing divided by the number of cells in the dimension concerned (i.e. the width). If the number of rows in Figure 3.29a were increased, as shown in Figure 3.29b containing two rows, then this would make no difference to the calculation of the processor weights, since a column of two cells would always be moved. The weight now refers to the time per group of cells, such as a column of cells, which is more often the case when dealing with structured mesh code problems. Unlike unstructured mesh code problems, where single cells can be moved, an entire row, column, or plane of cells is moved and so the weight no longer refers to a single cell.

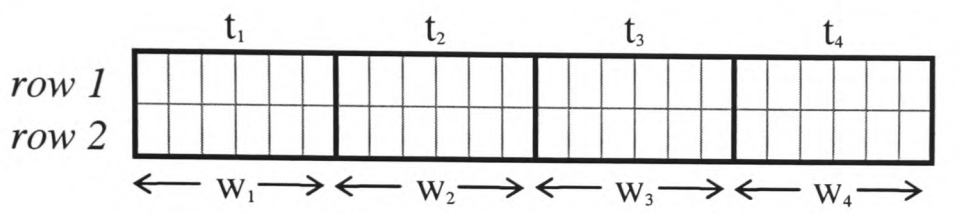
In Figure 3.29c there are four groups of processors containing a column of two processors each, where the timing for each of the eight processors is given along with the column width. In this instance an entire column of cells will be moved, irrelevant of who owns the row, and so the weight for each group is calculated representing the time per column of cells.

a)



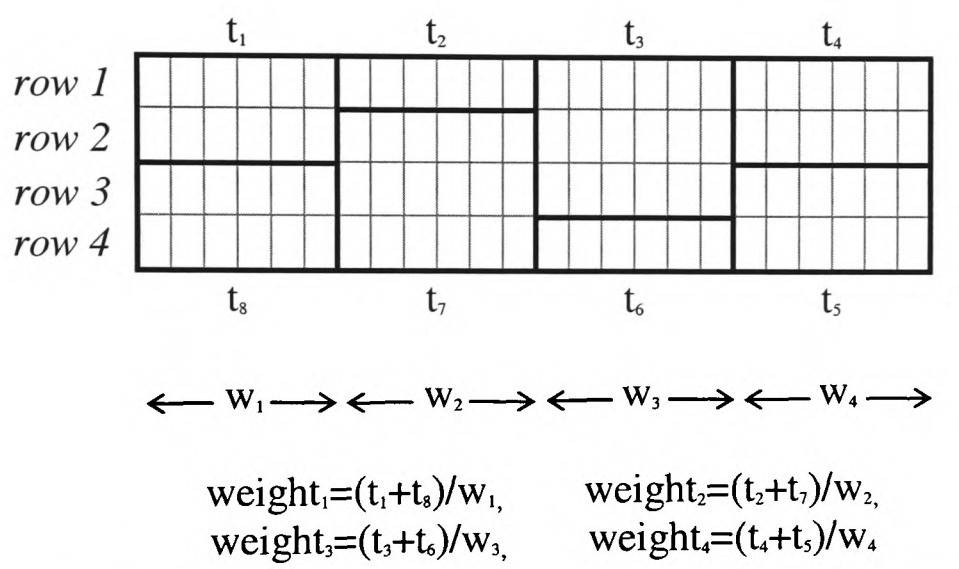
$weight_1=t_1/w_1$, $weight_2=t_2/w_2$, $weight_3=t_3/w_3$, $weight_4=t_4/w_4$

b)



$weight_1=t_1/w_1$, $weight_2=t_2/w_2$, $weight_3=t_3/w_3$, $weight_4=t_4/w_4$

c)



$weight_1=(t_1+t_8)/w_1$, $weight_2=(t_2+t_7)/w_2$,
$weight_3=(t_3+t_6)/w_3$, $weight_4=(t_4+t_5)/w_4$

Figure 3.29: Example showing a) a single row of cells that have been distributed onto 4 processors; b) two rows of cells that have been distributed onto 4 processors; and c) four rows of cells that have been distributed onto 8 processors (using a 4x2 topology). The weight (time to process a column of cells) can be calculated using t_p and w_p, representing the processor timing and width of cells on a processor.

The aim is to obtain the average weight on each processor (or group of processors), such that the workload is reduced on those processors with a high weight. If the workload is reduced on the processor with the maximum processor timing and placed onto other processors, then although the timing on the processor with the maximum timing will reduce, the timings of the other gaining processors will increase, particularly that of the processor with the minimum timing. It is unlikely that the timings will ever be reduced to that of the processor with the minimum timing, which is why it is more realistic to aim to reduce the timings to that of the average processor timing. Nevertheless, the overall execution time will be reduced from that of the processor with the maximum timing, improving the parallel performance.

It makes sense to move only boundary cells (cells adjacent to a neighbouring processor), as this will retain the rectangular partition and also simplify the communications needed to migrate data between processors (Section 2.6). The assumption that a processor processes any cell at its own rate therefore needs to be made. This assumption is precise for processor imbalance, as each cell on a processor takes the same time to compute.

A minimum and maximum restriction is placed on the amount of cells a processor owns, ensuring that the parallel code still operates correctly. During the parallelisation process the user is able to select the minimum width of the assignment region MIN_SLAB (Section B.9.1), dictating that every processor should have at least that many cells in the partitioned dimension. Communications updating the halo region only need to occur with immediately neighbouring processors, since these processors own the requested data. More communications would be needed in the parallel code if it were possible for a processor to own less than the minimum amount of cells, because the halo region would then need to be updated by a neighbour's neighbour as well, which is why this restriction is employed (Section 2.6). A restriction is also placed on the maximum amount of cells a processor can own due to memory constraints. The number of cells a processor can own is dependent upon the memory size, where it would be impossible to gain more cells than is physically possible. Memory reduction also needs to be considered for the same reason, where a processor can only gain as many cells that can fit into memory. It has been decided that a processor can only gain from, or lose to, an immediately neighbouring processor.

These restraints limit the extent of migration. For example, Processor 3 in Figure 3.30 can only gain cells from Processors 2 and 4, and alternatively it can only lose cells to these neighbouring processors. Therefore the maximum width on Processor 3 would be equal to the original width of Processor 3 plus the width of Processor 2 and Processor 4, minus twice the minimum width (i.e. 2xhalo width). In terms of Processor 3, the width of the halo region is left on its neighbouring processors (Processor 2 and 4) as a precaution, since it is not known with certainty that those neighbours will definitely gain cells from their other neighbours. For instance, it is possible for Processor 3 to gain all of Processor 4's cells if Processor 4 were to definitely gain the minimum width from Processor 5,

but this scenario cannot be guaranteed, in which case Processor 4 could end up with less than the minimum width.

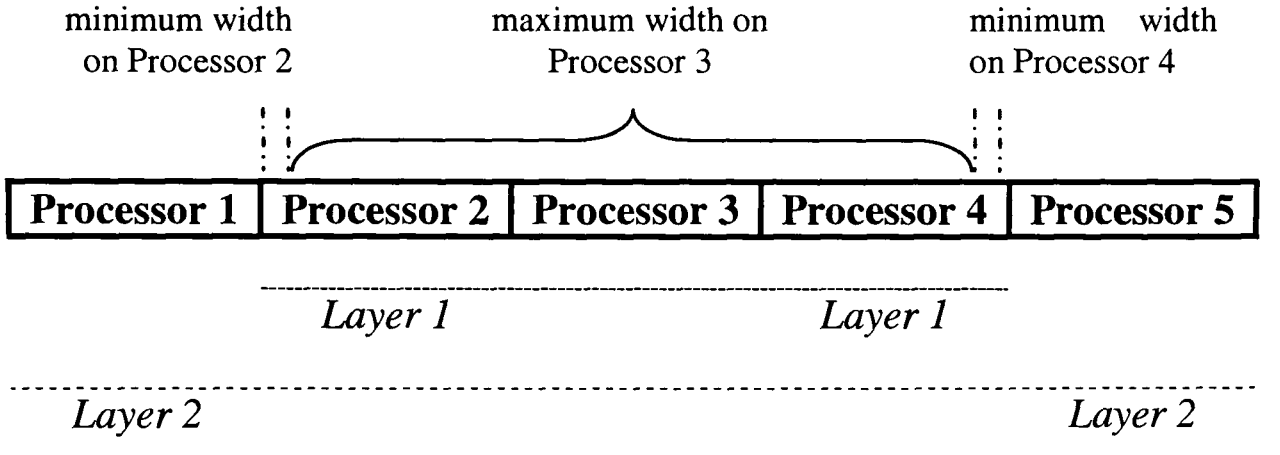


Figure 3.30: Processor 3 can only gain cells from, or lose cells to, its immediate neighbours (in Layer 1) Processor 2 and 4. The maximum number of cells that can be gained by Processor 3 is shown, taking into account the minimum width restriction on its neighbouring processors. Cells can be gained or lost to neighbours in Layer 2 in subsequent redistributions.

This calculation (Figure 3.31) illustrates a 'dampening' effect, where cells from the second layer of neighbours may be transferred in subsequent redistributions of the workload. For example, if it were desirable to transfer some cells from Processor 1 onto Processor 3, then those cells would first be transferred from Processor 1 onto Processor 2 in one load redistribution, and then from Processor 2 to Processor 3 in the next redistribution. The decision of whether or not to redistribute the workload is made every iteration, ensuring that the workload will be transferred at some point. Although a decision has been made that cells will only be transferred to a neighbouring processor, the situation in which a processor will be left with just the minimum amount will rarely occur. Most parallel problems are large enough so that when using a suitable number of processors each processor will have a sufficient workload (greater than the minimum amount).

```
C     Obtain the maximum width for Processor 1
      MAX_WIDTH(1)=WIDTH(1)+WIDTH(2)-MIN_WIDTH
C     Obtain the maximum width for intermediate processors
      DO I=2,N-1
        MAX_WIDTH(I)=WIDTH(I)+WIDTH(I-1)+WIDTH(I+1)-2*MIN_WIDTH
      END DO
C     Obtain the maximum width for Processor N
      MAX_WIDTH(N)=WIDTH(N)+WIDTH(N-1)-MIN_WIDTH
```

Figure 3.31: Calculation used to determine the maximum width on each processor.

The simple example shown in Figure 3.30 will be used to illustrate the algorithm used to calculate the new workload, where there are five processors with various amounts of cells for which new Left/Right limits need to be obtained. For simplicity only one row of processors has been used, meaning that the grouping of processors is not required for this example. The example shall be discussed in terms of processor imbalance meaning the assumption that each cell on a processor has the same weight is true. The case of physical imbalance shall be dealt with in Section 3.5.5.

The time to process the workload on each processor for a particular iteration is given in Table 3.5, along with the number of cells and the idle time. The maximum processor width is also given, where the minimum width (halo) is set to 1. There are 21 cells in total, where the average processor time is 230 seconds. It is obvious that Processor 2 is the slowest, taking twice as long as most of the other processors, and therefore needs to lose some of its load. The time per cell (processor weight) is also given, represented diagrammatically in Figure 3.32 that shall be used to illustrate the algorithm in more detail.

Processor	Number of cells	Time to process cells	Idle time	Weight	Maximum new width
1	4	160	290	40	4+5-1=8
2	5	450	0	90	5+4+4-2=11
3	4	240	210	60	4+5+5-2=12
4	5	150	300	30	5+4+3-2=10
5	3	150	300	50	3+5-1=7

Table 3.5: The number of cells on each processor in Figure 3.30 is shown, along with the time to process these cells (from which the idle time is calculated). The weight (time to process a cell) for each processor is given, along with the maximum width possible.

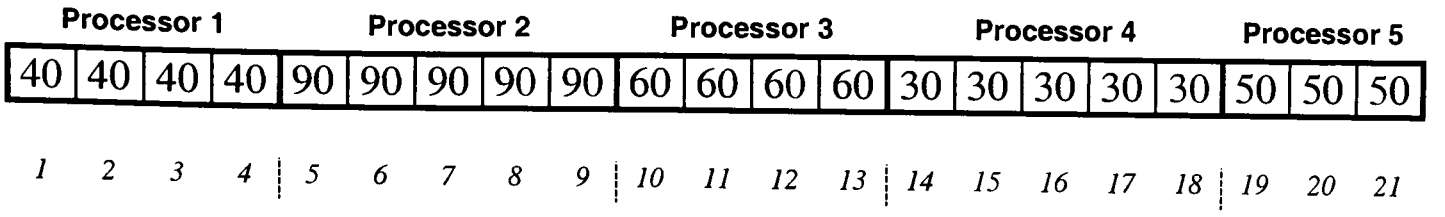


Figure 3.32: Graphical representation of the example shown in Figure 3.30, whose details are given in Table 3.5.

If distributed evenly, then ideally each processor should end up with a number of cells that is inversely proportional to its weight (w_i) meaning a processor whose weight is twice that of another processor should be allocated half as many cells. Therefore the number of cells allocated to each processor (n_i) is dependent on this proportion (f), as in Equation 3.1. The value of f can be obtained from the total number of cells (N) that need to be distributed where f=197.91 for the example shown in Figure 3.32.

$$n_i = \frac{f}{w_i}$$

$$N = \sum_{i=1}^{P} n_i = f \sum_{i=1}^{P} \frac{1}{w_i}$$

$$\therefore f = \frac{N}{\sum_{i=1}^{P} \frac{1}{w_i}}$$

Equation 3.1: Used to estimate the initial width on each processor when processor imbalance is presumed.

Given the initial width on each processor, an estimate of the initial distribution can be calculated (Figure 3.33) where the new width for a processor refers to the new load, and the current width refers to the original load. The initial estimate should return the number of cells a processor can process at their given weight, which is used as a basis to obtain the final redistribution of cells. The reason being that firstly, the new width may be less than the minimum width (i.e. the width of the halo region), in which case the new width needs to be increased to the minimum width. Secondly, for reasons that shall be made clearer when dealing with physical imbalance in Section 3.5.5, an upper limit is placed on the

new width so that a processor can only start off with their current width, enabling undistributed cells to then be reallocated. This is illustrated in Figure 3.34, where Processors 1, 4, and 5, start by owning all of their original cells, whereas Processor 2 starts with 2 of its own cells (180 seconds), and Processor 3 starts with 3 of its own cells (180 seconds). For example, the estimated width on Processor 4 is calculated as 197.91/30=6.6, which is reduced to 5 (its current width) since the estimated width exceeds its current width. Therefore 4 cells in total need to be reallocated onto neighbouring processors given this initial estimate.

```
C       Initial Width
        INITITAL WIDTH_P = f / WEIGHT_P
        IF(INITITAL WIDTH_P > CURRENT WIDTH_P )THEN
           NEW WIDTH_P = CURRENT WIDTH_P
        ELSE IF(INITITAL WIDTH_P < MINIMUM WIDTH )THEN
           NEW WIDTH_P = MINIMUM WIDTH
        ELSE
           NEW WIDTH = INITITAL WIDTH_P
        END IF
```

Figure 3.33: Calculation used to find the initial new distribution.

Initial Distribution:

Processor	Weight	Current Width	Current Time	Estimated Width	New Width	New Time
1	40	4	160	4.05	4	160
2	90	5	450	2.20	2	180
3	60	4	240	3.30	3	180
4	30	5	150	6.60	5	150
5	50	3	150	3.96	3	150

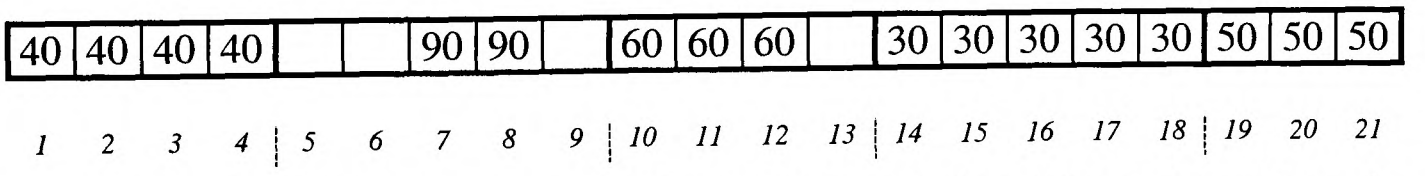


Figure 3.34: Graphical representation of the initial distribution of the problem shown in Figure 3.32 when processor imbalance is presumed.

After the initial distribution the undistributed workload can either be reallocated onto neighbouring processors, or the entire workload can be 'shifted' between neighbours, where there is a distinct difference. For example, in Figure 3.34 there are four undistributed cells (owned by Processors 2 and 3), meaning that it is not possible for Processor 5 to gain any additional cells if following the rule that undistributed cells can only be gained from a neighbouring processor. In

this example there are no cells on Processor 4 to be gained by Processor 5. Only Processors 1, 2, 3, and 4, can gain undistributed cells (where it is possible that Processors 2 and 3 may gain their own original cells).

If shifting the workload, then Processor 5 would be able to gain additional cells, allowing the workload on Processor 4 to shift to the Left if required. Although the reallocation method is valid, it does not allow cells to be filtered onto other processors apart from neighbouring processors and so it has been decided that the shift method shall be employed. If for instance Processor 5 in Figure 3.34 had only 2 cells, then it should gain cells, but this would only be possible if Processor 4 had some undistributed cells. With the shifting method, Processor 5 would be able to gain additional cells from Processor 4, who could gain cells from Processor 3 (shifting the workload onto more capable processors). To avoid load oscillation with the shift method, it has been decided that a processor may not gain cells from a processor to which it has already lost cells.

The undistributed cells are processed in an arbitrary order, as the allocated cell is deduced from the calculation to determine the gaining processor. In this example containing processor imbalance, a processor will gain additional cells at its own rate, which means that if Processor 1 were to gain an additional cell (given the initial distribution) then its new time would be 160+40=200 seconds. Even though Processor 1 is actually gaining a cell with a weight of 90 from Processor 2, the additional cell will be processed at a rate of 40. Similarly, if Processor 3 were to gain an additional cell then its new time would be 180+60=240, which just happens to be one of its own cells. Figure 3.35 shows the estimated processor timings if given an additional cell, where Processor 4 has an estimated time of 180 seconds. The undistributed cell should be allocated to the processor with the lowest estimated timing, where it is given to the processor with the smaller weight if several processors have the same timing. Therefore the first undistributed cell is allocated to Processor 4, whose width and timing are then increased to reflect the additional cell before proceeding to reallocate the remaining undistributed cells. Note that the estimated time can be calculated simply by multiplying the processor weight by the estimated width, but this is only true with processor imbalance.

Iteration 1:

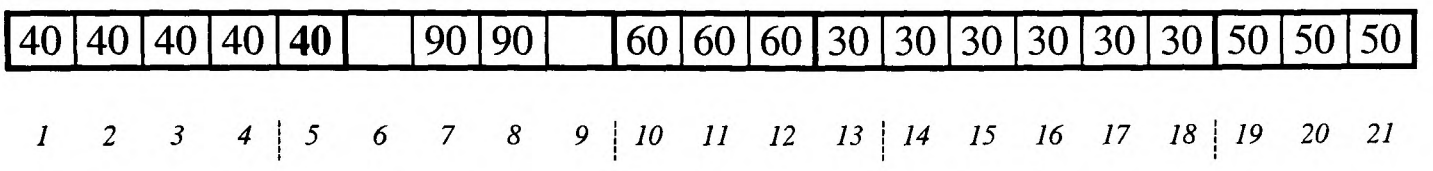
Processor	Weight	Width	Time	Estimated Width	Estimated Time	New Width	New Time
1	40	4	160	5	200	4	160
2	90	2	180	3	270	2	180
3	60	3	180	4	240	3	180
4	30	5	150	**6**	**180**	6	180
5	50	3	150	4	200	3	150

Figure 3.35: New distribution of the problem shown in Figure 3.32 after one iteration. The calculation of the estimated timing, if given an additional cell, is shown for each processor using the initial width and timings along with the processor weights. The additional cell is allocated to Processor 4 who has the lowest estimated timing.

The process is repeated until all of the undistributed cells have been reallocated. Figure 3.36 shows the current, estimated and new, width and timing for each processor, along with a graphical representation. It should be observed that in iteration 3 Processor 5 gains an additional cell from Processor 4, whose workload (and timing) is then decreased. If a processor gains a cell from a neighbour whose new width is less than their original width then this will not affect the neighbouring processor. The width and timing of the neighbouring processor will need to be reduced if the neighbours current width is greater than, or equal to, its original width. In this example (iteration 3), Processor 4 (the neighbouring processor that is losing a cell) currently has a new width (6) that exceeds its original width (5), meaning its current width and time will both have to be reduced when one of its cells is allocated to Processor 5.

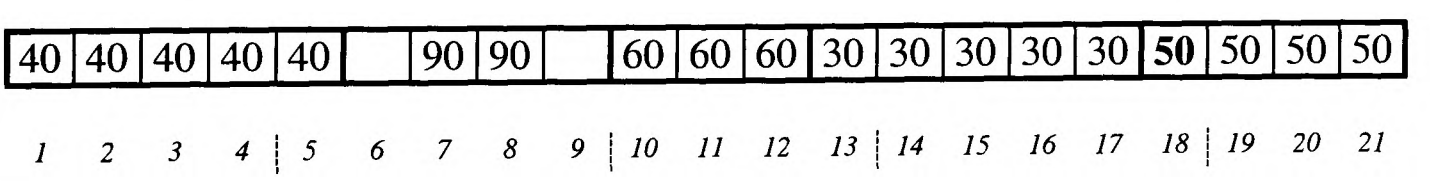
Iteration 2:

Processor	Weight	Width	Time	Estimated Width	Estimated Time	New Width	New Time
1	40	4	160	**5**	**200**	5	200
2	90	2	180	3	270	2	180
3	60	3	180	4	240	3	180
4	30	6	180	7	210	6	180
5	50	3	150	4	200	3	150



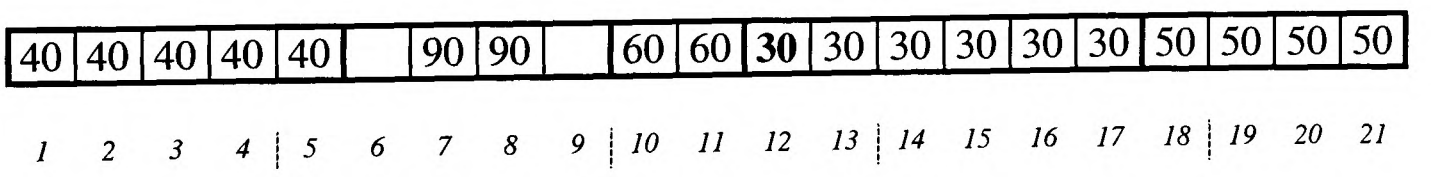
Iteration 3:

Processor	Weight	Width	Time	Estimated Width	Estimated Time	New Width	New Time
1	40	5	200	6	240	5	200
2	90	2	180	3	270	2	180
3	60	3	180	4	240	3	180
4	30	6	180	7	210	**5**	**150**
5	50	3	150	**4**	**200**	4	200



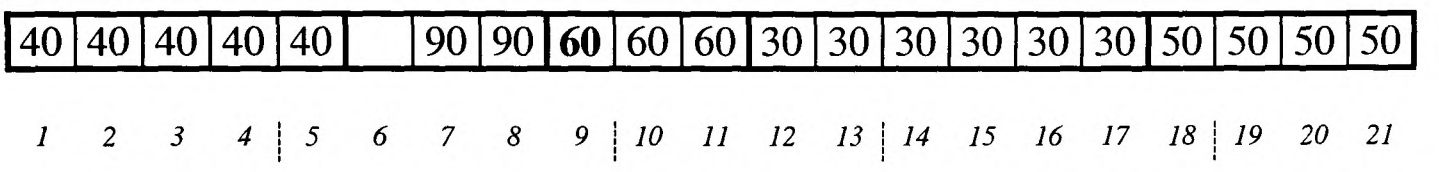
Iteration 4:

Processor	Weight	Width	Time	Estimated Width	Estimated Time	New Width	New Time
1	40	5	200	6	240	5	200
2	90	2	180	3	270	2	180
3	60	3	180	4	240	**2**	**120**
4	30	5	150	**6**	**180**	6	180
5	50	4	200	5	250	4	200



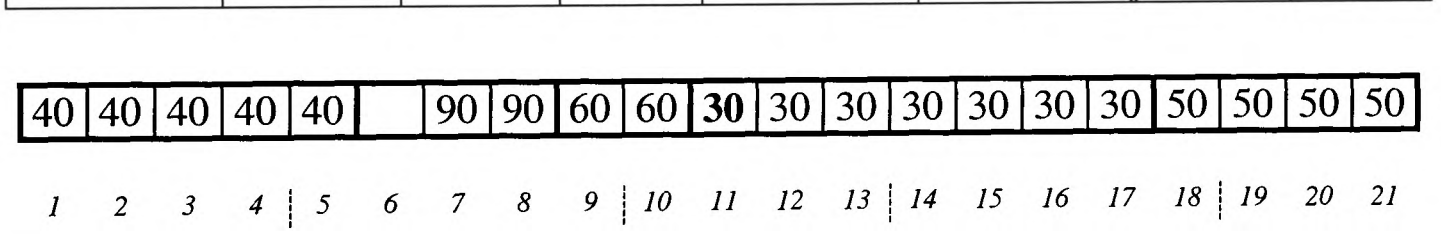
Iteration 5:

Processor	Weight	Width	Time	Estimated Width	Estimated Time	New Width	New Time
1	40	5	200	6	240	5	200
2	90	2	180	3	270	2	180
3	60	2	120	**3**	**180**	3	180
4	30	6	180	7	210	6	180
5	50	4	200	5	250	4	200



Iteration 6:

Processor	Weight	Width	Time	Estimated Width	Estimated Time	New Width	New Time
1	40	5	200	6	240	5	200
2	90	2	180	3	270	2	180
3	60	3	180	4	240	**2**	**120**
4	30	6	180	**7**	**210**	7	210
5	50	4	200	5	250	4	200



Iteration 7:

Processor	Weight	Width	Time	Estimated Width	Estimated Time	New Width	New Time
1	40	5	200	6	240	5	200
2	90	2	180	3	270	2	180
3	60	2	120	**3**	**180**	3	180
4	30	7	210	8	240	7	210
5	50	4	200	5	250	4	200

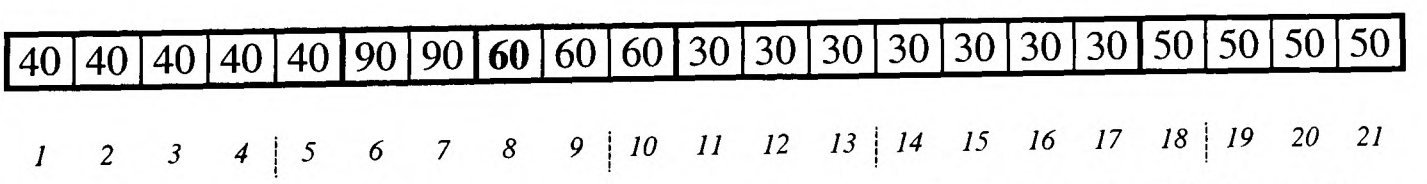


Figure 3.36: Several iterations that are used to find the new distribution of the problem shown in Figure 3.32.

The new distribution timings now appear to be balanced, where the average timing is now 190 seconds. The new load distribution is as expected with most cells on Processor 4 who had the lightest weight, and where Processor 2 has

the fewest cells. The new processor partition range limits can now be evaluated given the new processor workloads (i.e. the new widths).

3.5.2 Evaluating The New Processor Partition Range Limits

Once the new workload for each processor has been found it is then possible to evaluate the new processor partition range limits for each processor. In the above example the new limits are evaluated for Processor 1 (the leftmost processor) using its new width. The lower processor partition range limit for Processor 1 is 1, and its upper processor partition range limit is 1+5-1=5, implying that the new lower processor partition range limit for the next processor (Processor 2) is 6. The pseudo code to evaluate the new limits for this example is shown in Figure 3.37, where the new limits are actually calculated for a specific dimension (K) that is being balanced, enabling the algorithm to operate on any number of partitions. For instance, in this example the partition created on the first pass is being balanced where the new Left and Right limits are evaluated, which are represented within CAP_DLB_NEW_PROCLIMITS as 1 and 2 respectively. If the partition created on the second pass were being balanced then the Up and Down limits would be calculated using 3 and 4 within CAP_DLB_NEW_PROCLIMITS respectively. The old and new limits for this example can be seen in Table 3.6.

```
C       Initialise the lower limit of the first processor to be processed
        LOW=1
C       Evaluate the new limits for each processor in the balanced dimension K
C       If K=1 (partition created on 1st pass) then
C                       new Left (1) and Right (2) limits will be evaluated
C       If K=2 (partition created on 2nd pass) then
C                       new Up (3) and Down (4) limits will be evaluated
C       If K=3 (partition created on 3rd pass) then
C                       new Back (5) and Forth (6) limits will be evaluated
        DO PROC=1,5
C         Evaluate the new lower limit of the processor
          CAP_DLB_NEW_PROCLIMITS(PROC,(2*K)-1)=LOW
C         Evaluate the new upper limit of the processor using the
C         newly calculated width
          CAP_DLB_NEW_PROCLIMITS(PROC,2*K)=LOW+
     +                                   NEW_WIDTH(PROC)-1
C         Calculate the new lower limit for the next processor
          LOW=LOW+NEW_WIDTH(PROC)
        END DO
```

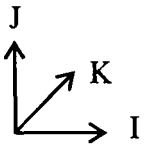
Figure 3.37: Pseudo code used to evaluate the new processor partition range limits for the processors in Figure 3.32.

	Old Limits		New Limits	
Processor	Lower (Left)	Upper (Right)	Lower (Left)	Upper (Right)
1	1	4	1	5
2	5	9	6	7
3	10	13	8	10
4	14	18	11	17
5	19	21	18	21

Table 3.6: The old and new processor partition range limits for the example shown in Figure 3.32.

As shown in Figure 3.29c, a 2D partition may be used where processors are grouped together (into columns of processors for instance). The new widths would be calculated for each group of processors, after which the new limits must be evaluated for all processors. For example, consider the case shown in Figure 3.38 that uses a 5x2x3 processor topology. In this example Processors 1, 10, 11, 20, 21, and 30 share the same Left/Right limits so that they can be grouped and treated as a single entity when calculating the new workloads. Similarly, the other processors can be grouped such that there are 5 entities (planes of processors) in total. If the widths and timings of these groups are the same as in the example Figure 3.32, then the same calculations will be made, leading to the groups having their corresponding new widths. Processors 1, 10, 11, 20, 21, and 30 will all have a new width of 5 in the first partitioned dimension (in the Left/Right direction). The pseudo code in Figure 3.37 therefore needs to be altered slightly to account

for the grouping of processors and this is shown in Figure 3.39. Table 3.7 shows the processor group timings along with the old and new widths for each group, and the new limits for this example.



21 (20)	**22** (40)	**23** (60)	**24** (35)	**25** (20)
30 (30)	**29** (100)	**28** (30)	**27** (40)	**26** (20)

11 (30)	**12** (50)	**13** (10)	**14** (25)	**15** (20)
20 (30)	**19** (100)	**18** (90)	**17** (10)	**16** (50)

1 (20)	**2** (70)	**3** (20)	**4** (20)	**5** (20)
10 (30)	**9** (90)	**8** (30)	**7** (20)	**6** (20)

Figure 3.38: Example of a 5x2x3 processor topology, where the processor numbers are given followed by the processor timing (in seconds) in brackets.

```
C       Initialise the lower limit of the first processor to be processed
        LOW=1
C       Evaluate the new limits for each processor in the balanced dimension K
        DO GROUP=1,5
          DO I=1,6
C           Process the I^th processor in the group
            PROC=FIND_PROC(I,GROUP)
C           Evaluate the new lower limit of the processor
            CAP_DLB_NEW_PROCLIMITS(PROC,(2*K)-1)=LOW
C           Evaluate the new upper limit of the processor using the
C           newly calculated group width
            CAP_DLB_NEW_PROCLIMITS(PROC,2*K)=LOW+
     +                                       NEW_WIDTH(GROUP)-1
          END DO
C         Calculate the new lower limit for the next group of processors
          LOW=LOW+NEW_WIDTH(GROUP)
        END DO
```

Figure 3.39: Amended pseudo code that is used to evaluate the new processor partition range limits for the groups of processors in Figure 3.38.

Group	Processors	Old Width	New Width	Timing	New Limits	
					Lower (Left)	Upper (Right)
1	1, 10, 11, 20, 21, 30	4	5	160	1	5
2	2, 9, 12, 19, 22, 29	5	2	450	6	7
3	3, 8, 13, 18, 23, 28	4	3	240	8	10
4	4, 7, 14, 17, 24, 27	5	7	150	11	17
5	5, 6, 15, 16, 25, 26	3	4	150	18	21

Table 3.7: Group widths (old and new), timings, and new processor partition range limits, for the processors in Figure 3.38.

3.5.3 Adjusting The Processor Timings

Having calculated the new processor partition range limits for one of the partitioned dimensions, the new processor partition range limits now have to be calculated for the subsequent partitioned dimensions. If there are several partitioned dimensions then, to account for the balance already obtained whilst processing this and previous dimensions, the processor timings must first be adjusted before processing the next partitioned dimension. For example, after calculating the new Left/Right limits for the problem shown in Figure 3.38, the new Up/Down limits (in a Non-Staggered Dimension) need to be calculated. If the original timings are used then this could lead to even more cells being shifted from the slow processors (in the Up/Down direction), which does not account for those cells already lost when balancing in the Left/Right direction. Adjusting the processor timings therefore has a 'damping' effect as fewer cells are moved than if the timings were not adjusted.

The processor timings must be adjusted to take into account the processor cells that are lost and gained. For demonstration purposes, consider Processor 4 in Figure 3.32 which originally had 5 cells taking 30 seconds each to process, giving an overall processor timing of 150 seconds. Note that the processor timings for this example would not actually need to be adjusted since there are no other

partitioned dimensions to process, this example is simply used to illustrate how the processor timings are adjusted. After redistributing the load, Processor 4 has lost 1 cell to Processor 5, and has gained 3 cells from Processor 3, meaning that its processor timing should be reduced by 30 seconds and increased by 90 seconds. With a new load of 7 cells, the adjusted timing of Processor 4 is 210 seconds, where each cell still takes 30 seconds to process. With processor imbalance it is known that every cell on a processor takes the same time to process. This means that a processor will lose and gain cells at its own rate such that the new load will all be processed at the original weight, therefore the calculation to adjust the timing for Processor 4 would be (150*7/5). Similarly, Processor 5 now has an adjusted timing of 200 seconds, and the other processors in the example have adjusted timings equal to those shown in the last iteration of Figure 3.36.

The example given above deals with a single line of processors (which is what happens when balancing the load in the Staggered Dimension), however when balancing in a Non-Staggered Dimension the timings need to be adjusted for each processor in every group. The new Left/Right limits for the 3D example in Figure 3.38 are evaluated to be the same as those for the 1D example in Figure 3.32, given the fact that the group widths and timings are the same as those shown in Figure 3.32. Therefore, to adjust the processor timings of Group 4, for instance, each processor timing in the group can be multiplied by 7/5 (giving a new overall group timing of 210 seconds). In general, the processor timings can be adjusted using:

$$\text{Adjusted Processor Timing} = \text{Old Timing} * (\text{New Width}/\text{Old Width})$$

where the adjusted timings (T') for the example in Figure 3.38 are given in Table 3.8 along with the original processor timings.

Group 1		
Proc	**T**	**T'**
1	20	25
10	30	37.5
11	30	37.5
20	30	37.5
21	20	25
20	30	37.5
	160	200

Group 2		
Proc	**T**	**T'**
2	70	28
9	90	36
12	50	20
19	100	40
22	40	16
29	100	40
	450	180

Group 3		
Proc	**T**	**T'**
3	20	15
8	30	22.5
13	10	7.5
18	90	67.5
23	60	45
28	30	22.5
	240	180

Group 4		
Proc	**T**	**T'**
4	20	28
7	20	28
14	25	35
17	10	14
24	35	49
27	40	56
	150	210

Group 5		
Proc	**T**	**T'**
5	20	26.7
6	20	26.7
15	20	26.7
16	50	66.7
25	20	26.7
26	20	26.7
	150	200

Table 3.8: The original processor timings (T) and the adjusted processor timings (T') are given for each of the 6 processors in the 5 groups.

3.5.4 Processing Subsequent Partitioned Dimensions

The next partitioned dimension can now be processed (balanced) having adjusted the processor timings. Continuing with the 3D example in Figure 3.38, the new Up/Down limits can be calculated using the adjusted timings (T') shown in Table 3.8. The same process to calculate the new workload is undertaken, this time grouping the processors according to their row position, such that each processor in a group has the same Up/Down limits. The processor group timings are given in Table 3.9, which are used with the group width (number of cells in the Up/Down direction) to calculate the new workload for each group. The new Up/Down limits can then be evaluated, after which the processor timings should be adjusted before processing the final partitioned dimension containing the Back/Forth limits (in the Staggered Dimension).

Group	Processors	Timing
1	1, 2, 3, 4, 5, 11, 12, 13, 14, 15, 21, 22, 23, 24, 25	411.1
2	10, 9, 8, 7, 6, 20, 19, 18, 17, 16, 30, 29, 28, 27, 26	559.1

Table 3.9: The group timings (using the adjusted processor times shown in Table 3.8) that are used to calculate the new workload in the Up/Down direction for the example given in Figure 3.38.

The processor partition range limits in the Staggered Dimension are non-coincidental, meaning that the processors are again grouped in a different manner. For example, if the Staggered Dimension in Figure 3.38 contains the Back/Forth limits, then there would be 10 groups of 3 processors (shown in Table 3.10), where each group is processed separately. The new Back/Forth limits are obtained for Processors 1, 11, and 21, in Group 1. Then the new Back/Forth limits are obtained for Processors 2, 12, and 22, in Group 2, and so on until the new limits are obtained for Group 10 containing Processors 10, 20, and 30. In this instance there is no need to adjust the processor timings, as no further partitioned dimensions are processed.

Group	Processors
1	1, 11, 21
2	2, 12, 22
3	3, 13, 23
4	4, 14, 24
5	5, 15, 25
6	6, 16, 26
7	7, 17, 27
8	8, 18, 28
9	9, 19, 29
10	10, 20, 30

Table 3.10: Processors would be grouped in 3's, where each of the 10 groups would be processed separately in the Staggered Dimension (dimension containing Back/Forth limits).

3.5.5 Processor Imbalance versus Physical Imbalance

The algorithm discussed above is suitable when processor imbalance is present, but the algorithm also needs to be able to deal with physical imbalance. With physical imbalance the physical characteristics of the application code make it

such that each cell on a processor takes a different time to compute, violating the assumption discussed above in Section 3.5.1. For example, consider the 5 cells on a processor shown in Figure 3.40, each taking a different time to compute (10, 6, 5, 3, and 1 seconds). It is difficult to identify the heavy cells when the individual cell timings on a processor are not evaluated, which is why it is easier to use the assumption that each cell takes 5 seconds to compute.

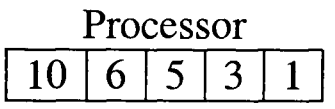


Figure 3.40: Simple example showing 5 cells on a processor, where the time to process each cell is different. Using the assumption that every cell on a processor takes the same time to compute, then each cell would take 5 seconds.

As with processor imbalance, each processor should ideally end up with roughly the same time, but unlike the initial distribution obtained with processor imbalance the new processor width is not inversely proportional to its weight. In this case, the cells weigh the same no matter where they are allocated, meaning that each processor should ideally end up with the average processor time. The initial width on each processor can be calculated as shown in Figure 3.41 (compare with Figure 3.33).

$$\text{INITIAL WIDTH}_P = \text{AVERAGE\_TIME} / \text{WEIGHT}_P$$

Figure 3.41: Estimate of the initial width on each processor when physical imbalance is presumed.

The number of cells a processor can process at its given weight, without exceeding the average time, can be calculated as shown, but this would be incorrect since additional cells would not necessarily be processed at the same weight. Additional cells would be processed at the weight of a neighbouring processor, that which has lost one of its cells.

Given the same distribution and timings as shown in Figure 3.32, but this time presuming physical imbalance, the initial distribution shown in Figure 3.42 can be calculated in a similar manner to that used with processor imbalance (see Figure 3.33), where each processor starts with a number of its own cells that can be processed at its own rate (weight). For example, the initial estimated width on

Processor 1 would be 230/40=5.75. Any cells not allocated at this stage now need to be reallocated onto neighbouring processors.

Initial Distribution:

Processor	Weight	Current Width	Current Time	Estimated Width	New Width	New Time
1	40	4	160	5.75	4	160
2	90	5	450	2.56	2	180
3	60	4	240	3.83	3	180
4	30	5	150	7.67	5	150
5	50	3	150	4.6	3	150

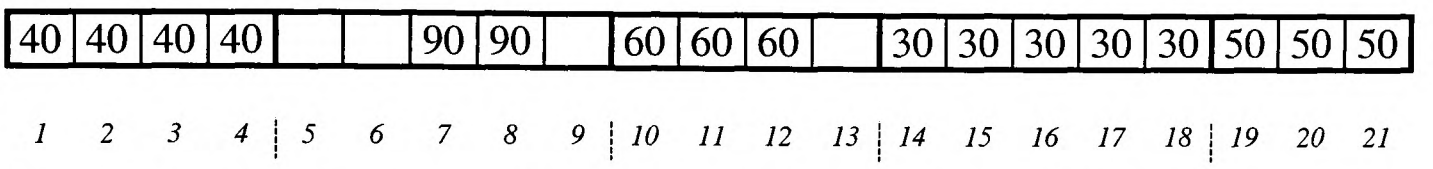


Figure 3.42: Graphical representation of the initial distribution of the problem shown in Figure 3.32 when physical imbalance is presumed.

Unlike processor imbalance, where any gained cells will be processed at a processor's own weight, whom a processor gains from is significant. If a processor still has unallocated cells on it (after the initial distribution), then any gained cells will be processed at the processor's own weight until all of its cells have been allocated. Any additional cells will then be processed at the weight of the losing processor, where a processor can either gain from their lower or upper neighbour. As before, an unallocated cell is given to the processor with the lowest estimated time. The processor timing is calculated if given either its own cell, or given a neighbouring cell from the lower or upper neighbour, i.e. a cell is gained either from the lower neighbour (L), from its self (S), or from the upper neighbour (U).

In Figure 3.43 Processor 1, who has no lower neighbour, is initially allocated all of its own cells, meaning that any additional cells can only be gained from its upper neighbour. Therefore the estimated time for Processor 1, given an additional cell, is its current time (160 seconds) plus the weight of Processor 2 (90 seconds), which equals 250 seconds. Not all of the cells on Processor 2 were initially allocated, implying that any additional cell will be gained from itself (S) and processed at its own weight, therefore having an estimated timing of 180+90=270 seconds. Similarly Processor 3 starts of with 1 unallocated cell,

meaning that the additional cell will be processed at its own weight, giving an estimated time of 180+60=240 seconds. All of the cells on Processor 4 were initially allocated, meaning that Processor 4 has an estimated width of 6 cells, where the additional cell can be gained from either of its neighbours. Two estimated timings are calculated for Processor 4, one if gaining an additional cell from its lower neighbour (L), and one if gaining an additional cell from its upper neighbour (U). Its current time (150 seconds) is increased by the weight of Processor 3 (60 seconds), giving an estimated time of 210 seconds. Its current time is also increased by the weight of Processor 5 (50 seconds), giving a different estimated time of 200 seconds. Processor 5, who has no upper neighbour, and who is initially allocated all of its own cells, can only gain additional cells from its lower neighbour (Processor 4) at a weight of 30 seconds, giving an estimated time of 150+30=180 seconds. The undistributed cell is allocated to the processor with the lowest estimated time, which in this instance is Processor 5, whose new width and time are set to the estimated width and time. The estimated timing of Processor 5 was calculated using the weight of the lower neighbour, indicating that the additional cell shall be gained from Processor 4. A point to note however is that there were no unallocated cells on Processor 4, meaning that a cell has actually been lost on Processor 4. The width and timing on Processor 4 therefore need to be amended before proceeding to distribute the remaining unallocated cells.

The same process is undertaken for the second iteration (and the remaining iterations, all shown in Figure 3.44), but this time Processor 4 cannot gain a cell from Processor 5 (its upper neighbour) to whom it has already lost a cell to. With 4 cells, at an adjusted time of 120 seconds, Processor 4 can now only gain an additional cell from Processor 3 (with a weight of 60 seconds). If it were to gain a cell from its Right, then that cell would be a cell that was previously lost, meaning that it would be processed at a weight of 30 seconds (its own weight). This would result in an estimated time of 150 seconds, which would be the lowest estimated timing in this iteration. Cell 18 would simply be shifted back onto Processor 4, and would oscillate between these two processors, rending the algorithm useless. For this reason, it has been decided that a processor will not be able to gain a cell from a processor to which it has already lost a cell.

For reasons associated with the migration of data (Section 2.6) a processor can only gain cells that were originally owned by its immediate neighbours. For example, Processor 4 can only gain cells originally owned by Processors 3 and 5, meaning that Processor 4 could own cells 10 to 21. It is obvious that Processor 4 should not gain all of these cells, as this may result in Processor 2 having no workload, and would definitely result in Processor 5 having no workload. Therefore to guarantee that each processor remains operational, each processor must operate on at least the minimum number of cells, i.e. the width of the halo regions. For example, if the minimum width is 1 then Processor 4 can gain cells 11 to 20, and still only need to communicate with Processors 3 and 5 to update its halo region. The number of cells that can be gained from a neighbour is therefore calculated (Figure 3.45) and used to limit the number of cells a processor can gain (comparable with Figure 3.31).

In iteration 8 for example, Processor 3 only has one of its original cells, where Processor 4 has gained the maximum number of cells (3) from its lower neighbour. Having already lost cells to Processor 5, Processor 4 can no longer be included in the calculation, as there are no more cells for it to gain.

Iteration 1:

Processor	Weight	Width	Time	Estimated Width	Estimated Time			New Width	New Time
					L	S	U		
1	40	4	160	5	-	-	250	4	160
2	90	2	180	3	-	270	-	2	180
3	60	3	180	4	-	240	-	3	180
4	30	5	150	6	210	-	200	**4**	**120**
5	50	3	150	**4**	**180**	-	-	4	180

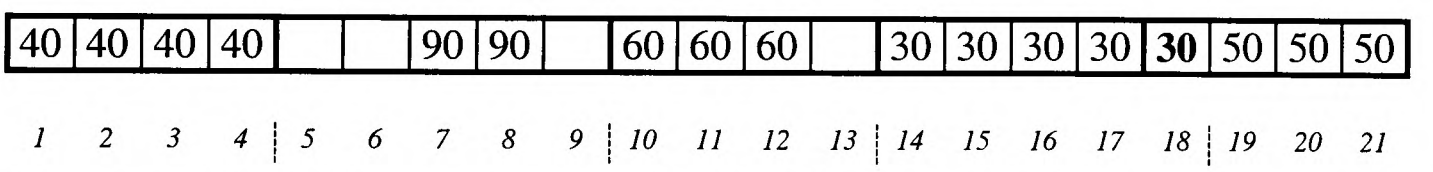
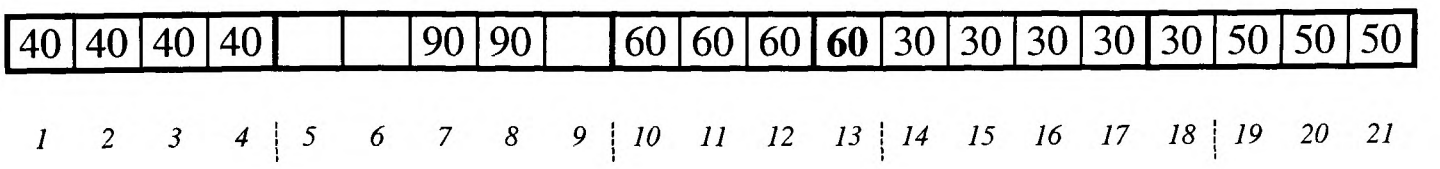


Figure 3.43: The 1st iteration (assuming physical imbalance) of the distribution of a cell in example Figure 3.42, given the initial distribution and the current processor widths and times. The estimated timing is calculated given the processor gains a cell from its lower neighbour (L), from its self (S), or from its upper neighbour (U), where possible.

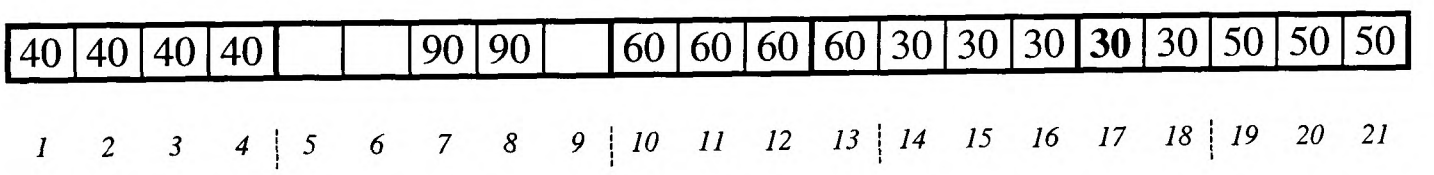
Iteration 2:

Processor	Weight	Width	Time	Estimated Width	Estimated Time			New Width	New Time
					L	S	U		
1	40	4	160	5	-	-	250	4	160
2	90	2	180	3	-	270	-	2	180
3	60	3	180	4	-	240	-	3	180
4	30	4	120	**5**	**180**	-	-	5	180
5	50	4	180	5	210	-	-	4	180



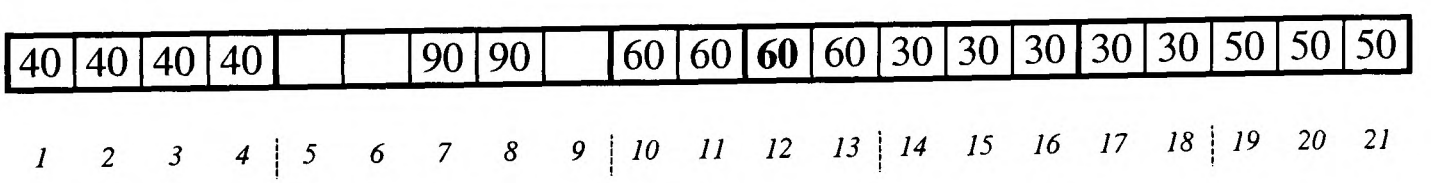
Iteration 3:

Processor	Weight	Width	Time	Estimated Width	Estimated Time			New Width	New Time
					L	S	U		
1	40	4	160	5	-	-	250	4	160
2	90	2	180	3	-	270	-	2	180
3	60	3	180	4	270	-	-	3	180
4	30	5	180	6	240	-	-	**4**	**150**
5	50	4	180	**5**	**210**	-	-	5	210



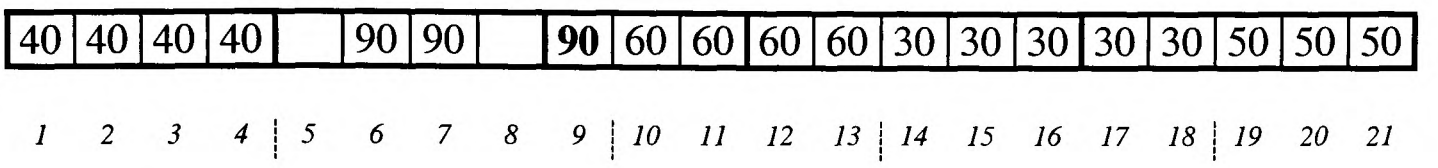
Iteration 4:

Processor	Weight	Width	Time	Estimated Width	Estimated Time			New Width	New Time
					L	S	U		
1	40	4	160	5	-	-	250	4	160
2	90	2	180	3	-	270	-	2	180
3	60	3	180	4	270	-	-	**2**	**120**
4	30	4	150	**5**	**210**	-	-	5	210
5	50	5	210	6	240	-	-	5	210



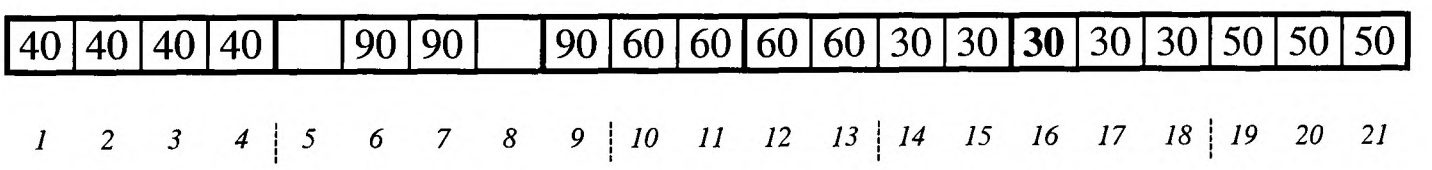
Iteration 5:

Processor	Weight	Width	Time	Estimated Width	Estimated Time			New Width	New Time
					L	S	U		
1	40	4	160	5	-	-	250	4	160
2	90	2	180	3	-	270	-	2	180
3	60	2	120	**3**	**210**	-	-	3	210
4	30	5	210	6	270	-	-	5	210
5	50	5	210	6	240	-	-	5	210



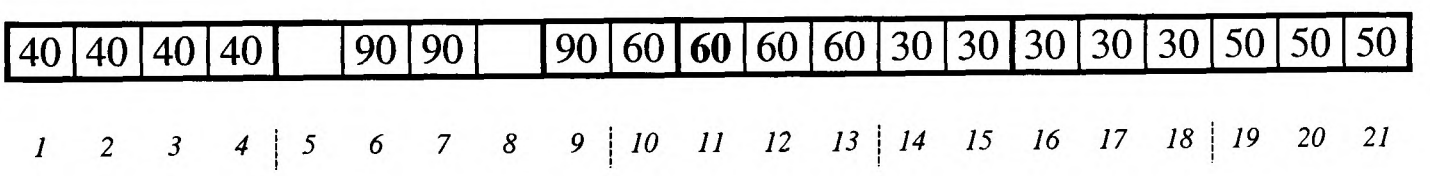
Iteration 6:

Processor	Weight	Width	Time	Estimated Width	Estimated Time			New Width	New Time
					L	S	U		
1	40	4	160	5	-	-	250	4	160
2	90	2	180	3	-	270	-	2	180
3	60	3	210	4	300	-	-	3	210
4	30	5	210	6	270	-	-	**4**	**180**
5	50	5	210	**6**	**240**	-	-	6	240



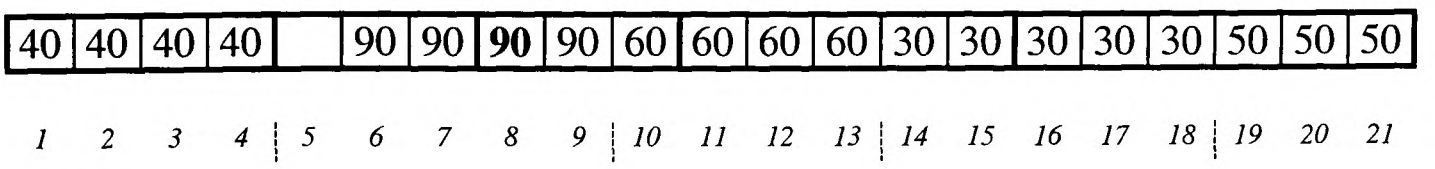
Iteration 7:

Processor	Weight	Width	Time	Estimated Width	Estimated Time			New Width	New Time
					L	S	U		
1	40	4	160	5	-	-	250	4	160
2	90	2	180	3	-	270	-	2	180
3	60	3	210	4	300	-	-	**2**	**150**
4	30	4	180	**5**	**240**	-	-	5	240
5	50	6	240	7	270	-	-	6	240



Iteration 8:

Processor	Weight	Width	Time	Estimated Width	Estimated Time			New Width	New Time
					L	S	U		
1	40	4	160	5	-	-	250	4	160
2	90	2	180	3	-	270	-	2	180
3	60	2	150	**3**	**240**	-	-	3	240
4	30	5	240	5	-	-	-	5	240
5	50	6	240	7	270	-	-	6	240



Iteration 9:

Processor	Weight	Width	Time	Estimated Width	Estimated Time			New Width	New Time
					L	S	U		
1	40	4	160	**5**	-	-	**250**	5	250
2	90	2	180	3	-	270	-	2	180
3	60	3	240	4	330	-	-	3	240
4	30	5	240	5	-	-	-	5	240
5	50	6	240	7	270	-	-	6	240

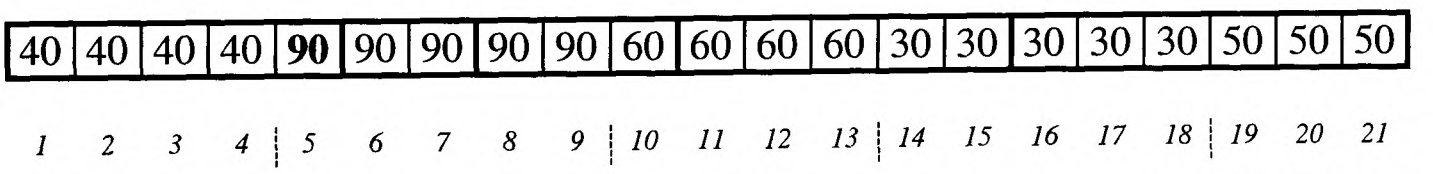


Figure 3.44: Remaining iterations that are used to redistribute the workload in example Figure 3.34 given that physical imbalance is assumed.

```
C     Calculate the number of upper (2) neighbouring cells to be
C     gained by Processor 1
      NEIGHB_NUM(1,1)=0
      NEIGHB_NUM(2,1)=WIDTH(2)-MIN_WIDTH
C     Calculated the number of neighbouring cells to be gained by
C     intermediate processors
      DO I=2,N-1
C        The lower neighbour
         NEIGHB_NUM(1,I)=WIDTH(I-1)-MIN_WIDTH
C        The upper neighbour
         NEIGHB_NUM(2,I)=WIDTH(I+1)-MIN_WIDTH
      END DO
C     Calculate the number of lower (1) neighbouring cells to be
C     gained by the last processor
      NEIGHB_NUM(1,N)=WIDTH(N-1)-MIN_WIDTH
      NEIGHB_NUM(2,N)=0
```

Figure 3.45: Pseudo code used to determine how many cells can be gained from a neighbouring processor, in the lower and upper direction, where the minimum width is equivalent to the width of the halo region.

Comparing the final distribution obtained with physical imbalance, shown in Figure 3.44, to that obtained with processor imbalance, shown in Figure 3.36, it is clear that there is a difference between the two types of problem. Both resulted in a reduced workload on Processor 2, who started with a time of 450 seconds, which is what should have happened. Bearing in mind that these are only estimates of the new processor timings, given that no other dimension needs to be balanced, the maximum processor timing is reduced to 210 seconds with processor imbalance, and to 250 seconds with physical imbalance. Note that the algorithm used to calculate the new workload for physical imbalance is also able to calculate the new workload for processor imbalance.

When processors are grouped together the new workload can be calculated in the same manner as with processor imbalance, where the new workload is calculated for each group rather than for a single processor. Similarly, the new processor partition range limits are evaluated in the same way. A difference exists in the way the processor timings are adjusted before processing another dimension, as it is no longer sufficient to calculate the new timing as a proportion of the original timing. Although cells are lost at a processor's own weight with processor imbalance, additional cells are not gained at a processor's own weight but at the weight of the losing neighbouring processor, as stated in Table 3.11.

	LOSE	GAIN
Processor imbalance	own weight	own weight
Physical imbalance	own weight	neighbour weight

Table 3.11: A cell will be lost at the weight of the current owner with both processor and physical imbalance, whereas a cell will be gained at the weight of the new owner with processor imbalance, and at the weight of the current owner with physical imbalance.

The pseudo code in Figure 3.46 can be used to adjust the original processor timings of the example in Figure 3.42. For the first processor (P=1), the new lower limit (NEW_LOW) is the same as the old lower limit (OLD_LOW), and so nothing is done in the first block of code. Although the new upper limit (NEW_HIGH) is greater than the old upper limit (OLD_HIGH), the adjustment for this processor will be made when dealing with Processor 2. Therefore when P=2, the new lower limit (5) is greater than the old lower limit (4), which means that 1 column of cells will be shifted, whose time is equal to 450*(1/5)=90

seconds. The current time on Processor 2 is therefore reduced by 90 seconds, which is placed onto the lower neighbour (Processor 1), giving Processor 1 an adjusted time of 250 seconds and Processor 2 an adjusted time of 360 seconds. The second condition is also true for Processor 2, as its new Right limit is less than its old Right limit. This time 2 columns of cells need to be shifted from Processor 2 onto Processor 3, with a time of 180 seconds, giving Processor 2 an adjusted time of 180 seconds and Processor 3 an adjusted time of 420 seconds. It can be seen that after adjusting all of the processor timings for the simple example above using the algorithm in Figure 3.46, that the adjusted processor timings equate to those calculated manually (shown as New Time in iteration 9).

```
C       Adjust timings for each processor
        DO P=1,5
C          Check lower limit
           IF (NEW_LOW(P) > OLD_LOW(P)) THEN
C             Calculate time to be shifted
              SHIFTED_TIME=ORIG_TIME(P)*((NEW_LOW(P)-OLD_LOW(P))/
     +                                                 ORIG_WIDTH(P))
C             Shift time off processor
              TIME(P)=TIME(P)-SHIFTED_TIME
C             Lower neighbour gains shifted time
              TIME(P-1)=TIME(P-1)+SHIFTED_TIME
           END IF
C          Check upper limit
           IF (NEW_HIGH(P) < OLD_HIGH(P)) THEN
C             Calculate time to be shifted
              SHIFTED_TIME=ORIG_TIME(P)*((OLD_HIGH(P)-NEW_HIGH(P))/
     +                                                 ORIG_WIDTH(P))
C             Shift time off processor
              TIME(P)=TIME(P)-SHIFTED_TIME
C             Upper neighbour gains shifted time
              TIME(P+1)=TIME(P+1)+SHIFTED_TIME
           END IF
        END DO
```

Figure 3.46: Pseudo code used to adjust the processor timings for the example in Figure 3.34.

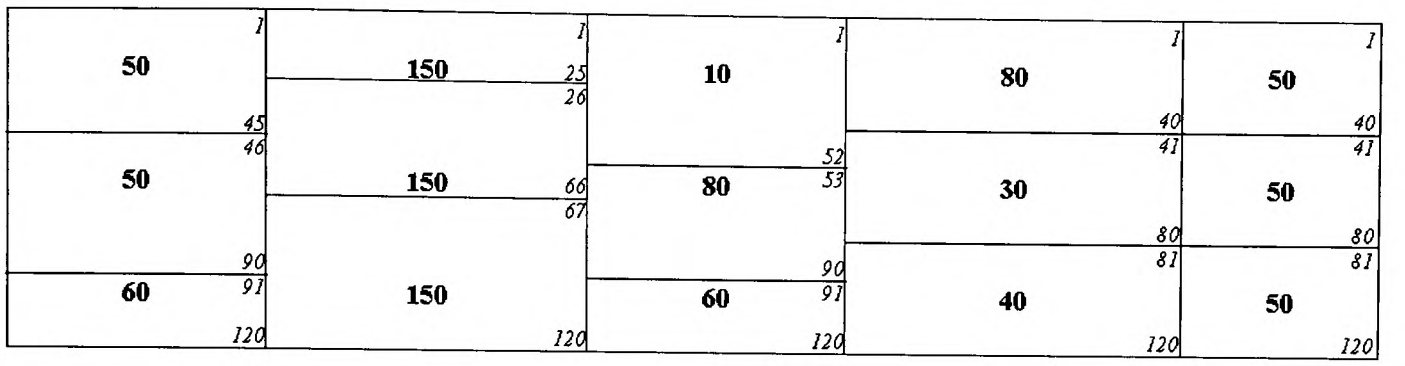
If the processors are grouped, such that in the above example there are 5 groups of 3 processors each (as shown in Figure 3.47), then the algorithm to adjust the processor timings needs to be changed to accommodate this. Note that the original width and timings of the 2D example in Figure 3.47 are the same as in the previous example. Using the new group widths (shown in iteration 9 of Figure 3.44) to find the new Left/Right limits, the processor timings need to be adjusted for each group. The new extended algorithm to adjust the processor timings, applied to this example, can be seen in Figure 3.48, where it is essential that the

staggered limits are taken into account when determining which neighbour provided the additional cells.

a) Processor topology

1	2	3	4	5
10	9	8	7	6
11	12	13	14	15

b) Processor timings and staggered limits



c) Group details

Group	Processors	Old Width	New Width	Timing
1	1, 10, 11	4	5	160
2	2, 9, 12	5	2	450
3	3, 8, 13	4	3	240
4	4, 7, 14	5	5	150
5	5, 6, 15	3	6	150

Figure 3.47: Example using a 5x3 processor topology, where the processor numbers are shown in a), and the processor timings and staggered limits are shown in b). The Group widths and timings are shown in c).

```
C       Adjust timings for the 3 processors in each of the 5 groups
        DO GROUP=1,5
          DO I=1,3
C           Identify the I^th processor in the group
            PROC=FIND_PROC(I,GROUP)
C           Check lower limit
            IF (NEW_LOW(PROC) > OLD_LOW(PROC)) THEN
C             Calculate time to be shifted – workload moved off this processor
              SHIFTED_TIME=ORIG_TIME(PROC)*((NEW_LOW(PROC)-
     +                                 OLD_LOW(PROC))/ORIG_WIDTH(PROC))
C             Shift time off this processor
              TIME(PROC)=TIME(PROC)-SHIFTED_TIME
C             Adjust the times of the 3 potential neighbours who will receive the
C             moved cells
              DO J=1,3
C               Identify the lower neighbouring processor in dimension K
C               E.g.: dealing with Left (1) neighbour if K=1
C                                  Up (3) neighbour if K=2
C                                  Back (5) neighbour if K=3
```

```
                NEIGH=PROC_NEIGHBOURS(J,(2*K)-1,PROC)
C               Check whether staggered limits of processor and neighbour
C               overlap
                IF (LOW_SD(NEIGH) ≤ HIGH_SD(PROC) and
     +                HIGH_SD(NEIGH) ≥ LOW_SD(PROC)) THEN
C                 Calculate overlap proportion
                  PROPORTION=MIN(HIGH_SD(PROC),HIGH_SD(NEIGH))
     +                          -MAX(LOW_SD(PROC),
     +                             LOW_SD(NEIGH))+1
C                 Lower neighbour gains proportion of the shifted time
                  TIME(NEIGH)=TIME(NEIGH)+SHIFTED_TIME*
     +                          (PROPORTION/WIDTH_SD(PROC))
                END IF
              END DO
            END IF
C           Check upper limit
            IF (NEW_HIGH(PROC) < OLD_HIGH(PROC)) THEN
C             Calculate time to be shifted – workload moved off this processor
              SHIFTED_TIME=ORIG_TIME(PROC)*((OLD_HIGH(PROC)-
     +                      NEW_HIGH(PROC))/ORIG_WIDTH(PROC))
C             Shift time off this processor
              TIME(PROC)=TIME(PROC)-SHIFTED_TIME
C             Adjust the times of the 3 potential neighbours who will receive the
C             moved cells
              DO J=1,3
C               Identify the upper neighbouring processor in dimension K
C               E.g.: dealing with Right (2) neighbour if K=1
C                                Down (4) neighbour if K=2
C                                Forth (6) neighbour if K=3
                NEIGH=PROC_NEIGHBOURS(J,2*K,PROC)
C               Check whether staggered limits of processor and neighbour
C               overlap
                IF (LOW_SD(NEIGH) ≤ HIGH_SD(PROC) and
     +                HIGH_SD(NEIGH) ≥ LOW_SD(PROC)) THEN
C                 Calculate overlap proportion
                  PROPORTION=MIN(HIGH_SD(PROC),HIGH_SD(NEIGH))
     +                          -MAX(LOW_SD(PROC),
     +                             LOW_SD(NEIGH))+1
C                 Upper neighbour gains proportion of the shifted time
                  TIME(NEIGH)=TIME(NEIGH)+SHIFTED_TIME*
     +                          (PROPORTION/WIDTH_SD(PROC))
                END IF
              END DO
            END IF
          END DO
        END DO
```

Figure 3.48: Amended pseudo code that is used to adjust the processor timings, which takes into account the grouping of processors and physical imbalance.

A processor can be identified by its position in the group, after which its limits can be tested. In this case the first processor in Group 1 is Processor 1, whose new Left limit is the same as before, and whose new Right limit is not less than before. Similarly, no operation is performed for the other two processors in Group 1 (Processor 10 and 11). When processing the second group, a reduction

needs to be made in both the lower and upper direction for Processors 2, 9, and 12. In this case 1 column of cells needs to be shifted in the lower direction for each processor, implying that the timing of Processor 2 will be reduced by 1/5 of the original processor timing (30 seconds), and similarly for Processor 9 and 12 in subsequent iterations of the I loop.

Having determined that the amount of time to be shifted off Processor 2 is 30 seconds, the next stage is to determine which processor receives this time. There are 3 potential neighbours in the lower direction (Processors 1, 10, and 11), each easily identified using an array which stores the neighbouring processors in every dimension (PROC_NEIGHBOURS). The Left and Right directions are represented using 1 and 2 respectively, and so the first Left neighbour of Processor 2 is PROC_NEIGHBOURS(1,(2*K)-1,2)=Processor 1 where K=1. Note that if the times were being adjusted after balancing the load in an Up/Down direction, then K would equal 2, and the directions of interest would be 3 (Up) and 4 (Down).

Having identified a neighbouring processor, the next step is to determine whether this processor needs to receive the shifted time. From Figure 3.47b it is clear that the column of cells on Processor 9 will only be gained by Processors 1 and 10, and not by Processor 11. A neighbouring processor will only gain cells if its staggered limits overlap with the staggered limits of the processor involved, and so this needs to be tested. If an overlap does occur, then this overlapping proportion of the shifted time is added to the neighbouring processor timing. For example, in this case the overlap region of Processor 1's staggered limits with Processor 2's staggered limits is 25/25 (i.e. 100% of Processor 2's loss), meaning that the time on Processor 1 will be increased by 30 seconds. A similar process happens with the upper limit of Processor 2, when a proportion of the original time is shifted onto the upper neighbours (Processors 3, 8, and 13), such that Processor 3 gains the cells (60 seconds) since it is the only overlapping neighbour.

When processing Processor 9 in Group 2, it is calculated that 30 seconds (1 column) need to be shifted to the Left. A proportion of this timing needs to shift onto Processor 1, and a proportion needs to shift onto Processor 10, where the proportions are 20/41 and 21/41 respectively given the staggered limits shown in Figure 3.47b. Similarly, the timing on Processor 9 will be reduced by another 60 seconds when dealing with the upper limit (on the Right), where the timing on

Processor 3 will increase by 39.51 seconds (60*27/41), and the timing on Processor 8 will increase by 20.49 seconds (60*14/41).

Had the timings been adjusted for the 3D example in Figure 3.38, then the number of groups would be 5 when dealing with the Left/Right direction (K=1), and the number of processors in each group would be 6. The number of potential neighbours would still be 3, since cells from a processor would be moved onto a processor in a different column (group) sharing the same Up/Down limits.

3.5.6 General Overview

The new partition range limits need to be calculated for each processor, in every dimension. The way in which the new limits are obtained can easily be altered (if desired) using a generic approach, where a change in the DLB utility will not affect the user's parallel application code. This allows the developer to test different approaches, which was deemed necessary when dealing with the different types of problems that can arise, such as when processor or physical imbalance occurs. Further investigation is still needed to deal with the case in which both types of load imbalance occur together in the same application code.

The call that calculates the new limits (CAP_DLB_FINDNEWLIMITS) is contained within another call (CAP_DLB_START_REBAL), which is only executed when load redistribution is required, hiding the operations of this utility away from the user.

As mentioned earlier in Section 2.1, the aim is to reduce the maximum processor time, as the overall execution time is dependent upon the slowest, or most heavily loaded processor. This means reducing the workload on those processors that are either 'slow' or 'heavy' (depending on whether the problem is said to have processor or physical imbalance). Therefore the new load can be calculated using the computation time (CAP_DLB_COMP_TIME) of each processor, which needs to be passed in since this is calculated externally.

3.6 Validate New Distribution

This utility is used to determine whether or not the load should be migrated. If enough cells are migrated then the new limits will be implemented, otherwise the current (old) limits will be used. The newly calculated distribution will not be implemented if the new distribution has not changed considerably from the current distribution.

A comparison of the new and old load is made for each processor, where the maximum difference is noted. If this maximum difference is greater than a user specified (or default) constraint then the load is said to be worth migrating, which is taken into account when comparing the limits in the next dimension to be analysed. The tolerance level can be set by the user, where the load will always be migrated no matter what if this is set to 1.

Figure 3.49 shows the code used to determine whether or not the newly calculated distribution should be implemented, where both the old and new limits in every dimension are stored internally. The number of cells on each processor is obtained using the old limits of that processor, which is compared to the new number of cells, which is calculated using the new limits of this and all previously migrated dimensions. If there are not enough cells to be migrated in a particular dimension then its limits are reset to the old limits.

```
      SUBROUTINE CAP_DLB_MIGRATE_RISK(
     +                                CAP_DLB_MIGRATE_DIM)
C     Declarations
C     Tolerance level – minimum amount of cells to be migrated in any given
C     dimension
      TOL=1
C     Work out how many cells are migrated in each dimension, taking the
C     migration of previous dimensions into account
      DO D=1,CAP_PROCDIM
        CAP_DLB_MIGRATE_DIMENSION(D)=.FALSE.
        MAX_CELLS_MIG=0
        DO I=1,CAP_NPROC
          OLD_NUMBER_OF_CELLS=1
          NEW_NUMBER_OF_CELLS =1
          DO J=1,CAP_PROCDIM
            L=(2*J)-1
            H=(2*J)
            IF( J.LT.D .AND. CAP_DLB_MIGRATE_DIMENSION(J) )THEN
C             Dimension J has already been migrated, number of cells
C             based on new width
              OLD_NUMBER_OF_CELLS =OLD_NUMBER_OF_CELLS*
     +                    (CAP_DLB_NEW_PROCLIMITS(H,I)-
```

```
     +                                  CAP_DLB_NEW_PROCLIMITS(L,I)+1)
                ELSE
C                  Dimension J has not been migrated, number of cells based on
C                  old width
                   OLD_NUMBER_OF_CELLS =OLD_NUMBER_OF_CELLS*
     +                           (CAP_DLB_PROCLIMITS(H,I)-
     +                             CAP_DLB_PROCLIMITS(L,I)+1)
                END IF
                IF( J.LE.D .AND. CAP_DLB_MIGRATE_DIMENSION(J) )THEN
C                  Dimension J has already been migrated, number of cells
C                  based on new width
                   NEW_NUMBER_OF_CELLS =NEW_NUMBER_OF_CELLS*
     +                           (CAP_DLB_NEW_PROCLIMITS(H,I)-
     +                             CAP_DLB_NEW_PROCLIMITS(L,I)+1)
                ELSE
C                  Dimension J has not been migrated, number of cells based on
C                  old width
                   NEW_NUMBER_OF_CELLS =NEW_NUMBER_OF_CELLS*
     +                           (CAP_DLB_PROCLIMITS(H,I)-
     +                             CAP_DLB_PROCLIMITS(L,I)+1)
                END IF
              END DO
C             Work out number of cells migrated to/from this processor
              CELLS_MIG=ABS(NEW_NUMBER_OF_CELLS-
     +                                        OLD_NUMBER_OF_CELLS)
C             Find the maximum number of cells migrated in this dimension
              IF( CELLS_MIG.GT.MAX_CELLS_MIG ) THEN
                MAX_CELLS_MIG=CELLS_MIG
              END IF
              END DO
              IF( MAX_CELLS_MIG.GE.TOL )THEN
C               Migrate data in this dimension
                CAP_DLB_MIGRATE_DIM(D)=.TRUE.
              ELSE
C               Reset processor limits – do not migrate in this dimension
                DO I=1,CAP_NPROC
                  CAP_DLB_NEW_PROCLIMITS((2*D)-1,I)=
     +                                  CAP_DLB_PROCLIMITS((2*D)-1,I)
                  CAP_DLB_NEW_PROCLIMITS((2*D),I)=
     +                                  CAP_DLB_PROCLIMITS((2*D),I)
                END DO
              END IF
            END DO
```

Figure 3.49: Utility used to determine whether or not to actually implement the newly calculated distribution. Migrate data in dimension only if enough cells are migrated in this dimension.

3.7 Migrating Data To Satisfy The New Partition

In order to implement the new distribution of this DLB strategy, the data needs to be migrated onto neighbouring processors to ensure correct processor ownership of up-to-date data values. This essentially means communicating data onto the

new owner of the data, where the new processor partition range limits can be used to determine where to place the migrated data. Potentially hundreds of variables could be migrated, where several communications are necessary to ensure that a single variable is migrated correctly, which could appear obtrusive, cluttering the user's code. The following generic utilities are therefore used in an attempt to minimise the changes to the user's code, where the load is migrated in each of the partitioned dimensions using the new processor limits of the previously migrated dimensions. This stage needs to be efficient since the time spent at this stage should not overshadow the parallel execution. Data should not be moved unnecessarily, but each processor must own the values of the data allocated to it after redistribution.

The direction, start address, and amount of data to be migrated, will differ for each variable in each redistribution, where several communication messages may be needed when migrating in a Non-Staggered Dimension (for each of the neighbours). Therefore two migration calls are needed, whereby the parameters of the call are used to determine the internal communication call used to migrate the load in either the Staggered or Non-Staggered Dimension (CAP_MIGRATE and CAP_DLB_MIGRATE respectively). It is possible to use CAP_MIGRATE when migrating in a Non-Staggered Dimension, if the migrated data is not also partitioned in the Staggered Dimension, suggesting the need to know the dimension in which the data is being migrated. The processor axes number (IAXES), indicating on which pass this dimension was partitioned, is therefore passed into the migration utility as MD, (Migration Dimension).

The basic components of a communication call (Section A.3.3), the start address, message length, data type, and communication direction, are essential to the functionality of the migration utilities, since the underlying operation is a communication call. These parameters therefore need to be passed into the migration utility as shown in Figure 3.50. Like the DLB communications discussed in Section 3.3, the original communication will be manipulated, such that the new start address, new message length, and with whom to communicate, are determined within the call at runtime.

```
CAP_MIGRATE      (A,START_IND,STRIDE,
                  S1,NS1,S2,NS2,S3,NS3,S4,NS4,S5,NS5,S6,NS6,
                  ITYPE,MD)

CAP_DLB_MIGRATE  (A,START_IND,STRIDE,STAG_IND,STAG_STRIDE,
                  S1,NS1,S2,NS2,S3,NS3,S4,NS4,S5,NS5,S6,NS6,
                  ITYPE,MD)
```

Figure 3.50: The migration utilities that are used to migrate data in the Staggered Dimension (CAP_MIGRATE), and in a Non-Staggered Dimension (CAP_DLB_MIGRATE).

3.7.1 Starting Address Of The Migrated Data

The starting address of the migrated data (A) is specified as being some location in memory from which the internal communications will be offset. The low declared limit is naturally used for all of the unpartitioned indices, as the entire range of this unpartitioned dimension will be migrated, but this is not the case for the partitioned indices. The lower processor partition range limit is specified for all Non-Staggered Dimensions excluding the Migration Dimension, since it is only necessary to migrate the data between the relevant processor partition range limits. The low declared limit is specified for both the Migration Dimension and the Staggered Dimension (these refer to the same index if using CAP_MIGRATE), as the new internal starting address will be offset from these indices.

Figure 3.51 demonstrates how to construct the starting address for the variable T (that has 7 dimensions), which is partitioned as shown. The starting address of T uses the low declared limit for all unpartitioned indices and also for index 3 and 1 (the Migration Dimension index and the Staggered Dimension index), when migrating in the Left/Right direction (MD=1). The lower processor partition range limit (CAP2_LOW) is used for the Non-Staggered Dimension that was partitioned second, where the old (current) value of CAP2_LOW is used in index 6. The processor partition range limits of Migration Dimension 1 will be updated after migrating all of the data in this dimension, meaning that the new values of CAP1_LOW and CAP1_HIGH will be used in subsequent code.

The starting address of T when migrating in the Up/Down direction (pass 2) again uses the low declared limit for the unpartitioned indices, the Migration

Dimension index, and the Staggered Dimension index. Similarly, the lower processor partition range limit is used for all of the Non-Staggered Dimensions excluding the Migration Dimension, which is the first partitioned dimension in this example (CAP1_LOW). The new value of CAP1_LOW is used in index 3 having already migrated the data in dimension 1.

When migrating in the last partitioned dimension (the Staggered Dimension), the new processor partition range limits are used for all of the Non-Staggered Dimensions (CAP1_LOW and CAP2_LOW in this example). In the same way, the low declared limit is used for the Staggered Dimension index (index 1) as well as for the unpartitioned indices. It is apparent that the starting address is different for the three Migration Dimensions, which is a reason why three calls are used to migrate the data rather than using a single call. If a single call were used to migrate T, then the construction of that call would become complicated (especially with all of the other parameters needed to migrate the data in a particular dimension). Additionally, not all of the arrays would need to be migrated in every dimension, since they may not be partitioned in every dimension.

T(**5:10**, 3:9, **4:8**, 7:14, 15:23, **20:23**, 1:3)

INDEX	PASS	STRIDE	LIMITS
1	**3**	**1**	**CAP3_LOW/CAP3_HIGH**
2		1x6=6	
3	**1**	**1x6x7=42**	**CAP1_LOW/CAP1_HIGH**
4		1x6x7x5=210	
5		1x6x7x5x8=1680	
6	**2**	**1x6x7x5x8x9=15120**	**CAP2_LOW/CAP2_HIGH**
7		1x6x7x5x8x9x4=60480	

Initial starting address for migration call: T(5,3,4,7,15,20,1) with partitioning: T(CAP3_LOW,3,CAP1_LOW,7,15,CAP2_LOW,1)
The actual starting address of the migration call may be lower or higher than the current processor partition range limits, and so it is calculated internally based on the low declared limit.

Migration Dimension	Migration call parameters
1	cap_dlb_migrate(T(**5**,3,**4**,7,15,**CAP2_LOW**,1),...,1)
2	cap_dlb_migrate(T(**5**,3,**CAP1_LOW**,7,15,**20**,1),...,2)
3	cap_migrate(T(**5**,3,**CAP1_LOW**,7,15,**CAP2_LOW**,1),...,3)

Figure 3.51: Example illustrating the starting address of an array to be migrated, in which the low declared limit is used in all but the Staggered Dimension and the Migration Dimension.

3.7.2 Starting Index And Stride Of The Migration And Staggered Dimensions

As with the DLB communications (Section 3.3.1.1), the starting index (START_IND) and stride (STRIDE) of the Migration Dimension need to be passed into the migration utility so that the new starting address in the Migration Dimension can be determined, as this differs for each partitioned index of every migrated variable. An address from which to physically migrate the data shall be obtained based on the starting address passed into the utility. The starting address shall be offset by a number of STRIDES from the START_IND of the Migration Dimension. These parameters need to be passed into the utility since there is no other way of knowing their values.

The index and stride of the Migration Dimension are used for START_IND and STRIDE respectively when using CAP_DLB_MIGRATE or CAP_MIGRATE. Continuing with the example shown in Figure 3.51, Figure 3.52 illustrates the values of START_IND and STRIDE for each of the Migration Dimensions.

Migration Dimension	Migration call parameters
1	cap_dlb_migrate(T(5,3,4,7,15,CAP2_LOW,1),**4,42**,...,1)
2	cap_dlb_migrate(T(5,3,CAP1_LOW,7,15,20,1),**20,15120**,...,2)
3	cap_migrate(T(5,3,CAP1_LOW,7,15,CAP2_LOW,1),**5,1**,...,3)

Figure 3.52: Example showing how to construct the START_IND and STRIDE parameters for the Migration Dimension.

When migrating in a Non-Staggered Dimension, using CAP_DLB_MIGRATE, each processor may have to communicate with several neighbours, in which case the starting address also needs to be offset in the Staggered Dimension. Therefore the starting index (STAG_IND) and stride (STAG_STRIDE) of the Staggered Dimension need to be passed into the CAP_DLB_MIGRATE utility, as illustrated in Figure 3.53.

Migration Dimension	Migration call parameters
1	cap_dlb_migrate(T(5,3,4,7,15,CAP2_LOW,1),4,42,**5,1**,...,1)
2	cap_dlb_migrate(T(5,3,CAP1_LOW,7,15,20,1),20,15120,**5,1**,...,2)
3	cap_migrate(T(5,3,CAP1_LOW,7,15,CAP2_LOW,1),5,1,...,3)

Figure 3.53: Example showing how to construct the STAG_IND and STAG_STRIDE parameters for calls to CAP_DLB_MIGRATE.

3.7.3 Migration Length

Currently, data only needs to be communicated over one buffered dimension, which is sufficiently handled within the buffered communications (Section A.3.3.3) of CAPTools. However, buffering over several dimensions may be necessary when redistributing data as planes of data may be moved for example. The dimensionality of each variable may differ from code to code, which means

that there is no set number of dimensions to buffer. Instead of looping through the various dimensions of the variable, the data should be buffered internally within the migration call, maintaining simplicity in the user's code. Therefore, like the buffered DLB communications mentioned in Section 3.3.2, the stride (S_i) and number of strides (NS_i) are needed to buffer the data for each dimension (i) of the variable being migrated. Note that the term paired-index is often used to describe both S and NS which define either an index of a group of contiguous indices. Apart from the Migration and Staggered Dimensions, all other dimensions need to be included in the migration length, since the former two dimensions are catered for in the previous parameters (see previous Section), and illustrated in Figure 3.54. If there are more parameters than there are dimensions of the migrated data, then dummy values are used for those parameters (i.e. S and NS are set to 1).

Migration Dimension	Migration call parameters
1	cap_dlb_migrate(T(5,3,4,7,15,CAP2_LOW,1),4,42,5,1,**6,7,210,8, 1680,9,15120,CAP2_HIGH-CAP2_LOW+1, 60480,3,1,1**,...,1)
2	cap_dlb_migrate(T(5,3,CAP1_LOW,7,15,20,1),20,15120,5,1,**6,7, 42,CAP1_HIGH-CAP1_LOW+1,210,8,1680, 9,60480,3,1,1**,...,2)
3	cap_migrate(T(5,3,CAP1_LOW,7,15,CAP2_LOW,1),5,1,**6,7, 42,CAP1_HIGH-CAP1_LOW+1,210,8,1680,9, 15120,CAP2_HIGH-CAP2_LOW+1,60480,3**,...,3)

Figure 3.54: Example showing how to construct the S and NS parameters representing the migration length.

Since the number of dimensions which need buffering is unknown, and can vary, it has been decided that up to six dimensions can be buffered inside a migration call, not including the Migration Dimension or the Staggered Dimension. It is unusual for a variable to have more than seven dimensions, and so this figure should be sufficient (it would not be difficult to change this, if required). Note that if data is to be 'packed' into a buffer (using multi-buffering), then that data will also need to be 'unpacked' from the buffer (using multi-unbuffering).

3.7.4 Type Of Data Being Migrated

The type of data being communicated (ITYPE) is an integral part of the migration utilities, enabling the utilities to operate generically on any type of data by converting the data into bytes using CAP_TYPELENS (Figure 3.55). It is used to find the new starting address, where the internal communications of the migration utility will then use CAP_BYTE instead of ITYPE. For example, if T was a REAL variable, then ITYPE would be passed in as 2 (since this type of data is stored within CAPTools as 2, compared with 1 for INTEGER type variables).

Migration Dimension	Migration call parameters
1	cap_dlb_migrate(T(5,3,4,7,15,CAP2_LOW,1),4,42,5,1,6,7,210,8, 1680,9,15120,CAP2_HIGH-CAP2_LOW+1, 60480,3,1,1,**2**,1)
2	cap_dlb_migrate(T(5,3,CAP1_LOW,7,15,20,1),20,15120,5,1,6,7, 42,CAP1_HIGH-CAP1_LOW+1,210,8,1680,9, 60480,3,1,1,**2**,2)
3	cap_migrate(T(5,3,CAP1_LOW,7,15,CAP2_LOW,1),5,1,6,7, 42,CAP1_HIGH-CAP1_LOW+1,210,8,1680,9, 15120,CAP2_HIGH-CAP2_LOW+1,60480,3,**2**,3)

Figure 3.55: Example showing how to construct the ITYPE parameter (where 2 is used to represent data of type REAL).

3.7.5 The Load Migration Algorithm For CAP_MIGRATE

The load is migrated in a particular dimension if there is a difference between the old and the new processor partition range limits of that dimension. Both the lower and the upper limits need to be compared, where it is possible that the data may have to be migrated in the lower and upper direction. The old and new limits are stored internally (see Section 3.2.2 and Section 3.5), and can be retrieved using the functions OLDLIMIT and NEWLIMIT, as seen in Figure 3.56. A call to these functions will return the limits for the calling processor (where CAP_PROCNUM identifies the calling processor). If LIM=1 then the lower processor partition range limit will be extracted, whereas the upper processor partition range limit will be extracted if LIM=2. The processor partition range limit that is returned is

dependent on the specified dimension (D), which is the Migration Dimension when called from the migration utilities. If D=1 then either the Left or Right limit is returned, as these were created on the first pass, if D=2 then either the Up or Down limit is returned, or if D=3 then the Back or Forth limit is returned.

```
      INTEGER FUNCTION OLDLIMIT(D,LIM)
      OLDLIMIT=CAP_DLB_PROCLIMITS((2*D)-(2-LIM),
     +                                          CAP_PROCNUM)


      INTEGER FUNCTION NEWLIMIT(D,LIM)
      NEWLIMIT= CAP_DLB_NEW_PROCLIMITS((2*D)-(2-LIM),
     +                                          CAP_PROCNUM)
```

Figure 3.56: Return the old or new processor partition range limit in either the lower (LIM=1) or upper (LIM=2) direction of the partitioned dimension D for the calling processor (CAP_PROCNUM).

As demonstrated in Figure 3.57, if the new lower limit is less than the old lower limit (L_x), then the data needs to be received from the lower direction, starting from the new lower limit (L_x). Alternatively, if the new lower limit is higher than the old lower limit, then the data needs to be sent in the lower direction, starting from the old lower limit (H_x). Similarly, if the new upper limit (H_x) is greater than the old upper limit, then the data needs to be received from the upper direction, starting from the old upper halo region ($H_x+1=L_{x+1}$). Additionally, the data needs to be sent in the upper direction, starting from the new upper halo region ($H_x+1=L_{x+1}$), if the new upper limit is smaller than the old upper limit. Figure 3.58 illustrates how to determine the amount to migrate (SECTION) and from where to begin migrating (START) for the Migration Dimension.

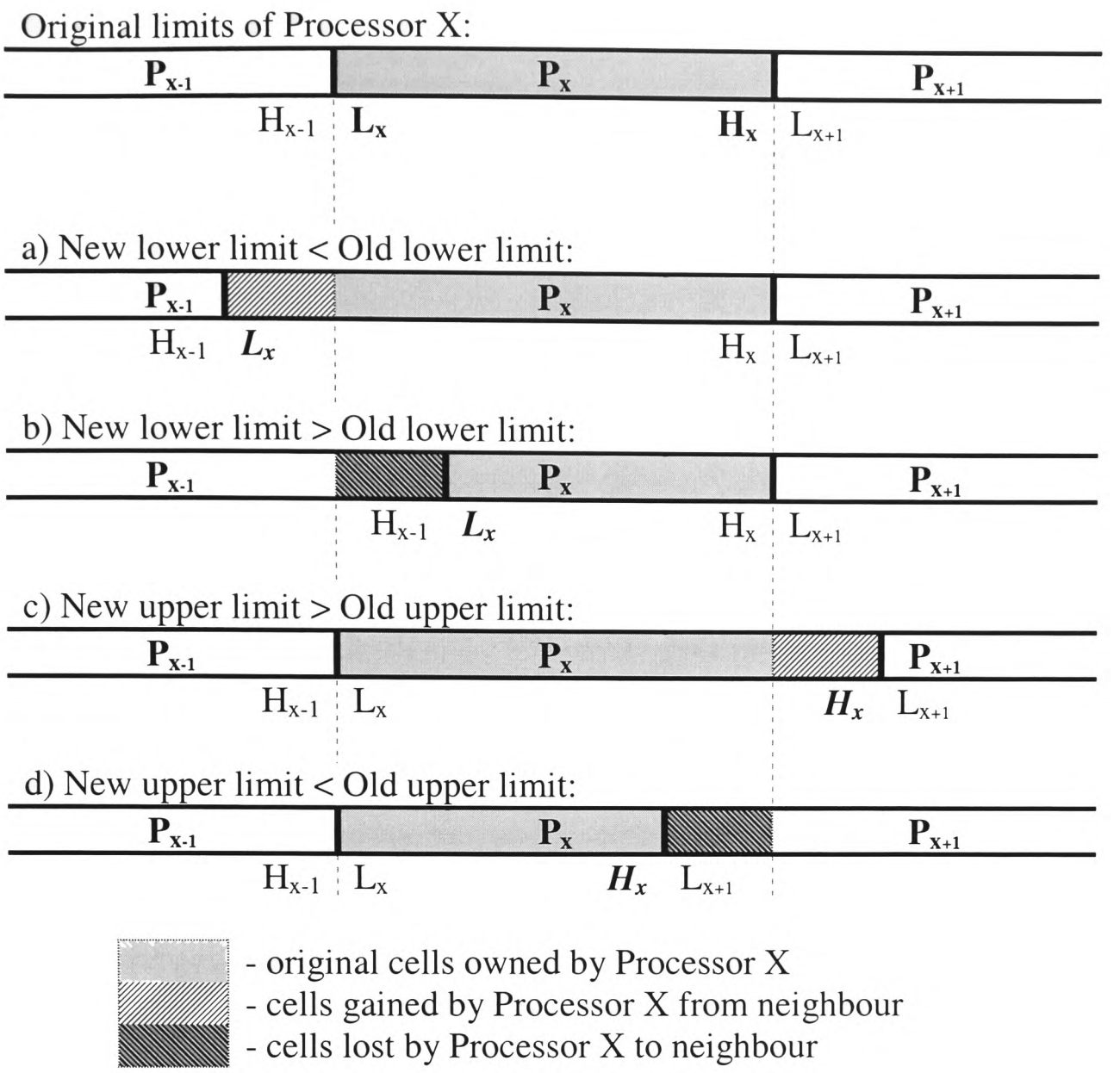


Figure 3.57: Example illustrating various situations after load redistribution for Processor X whose old lower and upper limits are represented by L_x and H_x respectively, and whose new lower and upper limits are represented by L_x and H_x respectively.

```
C       Obtain the old and new lower limit in the Migration Dimension (MD) for
C       the migrating processor
        OLDLOW=OLDLIMIT(MD,1)
        NEWLOW=NEWLIMIT(MD,1)
C       Examine the lower limits of migrating processor
        IF( NEWLOW.LT.OLDLOW )THEN
C          Need to receive data in lower direction
           START=NEWLOW
           SECTION=OLDLOW-NEWLOW
           ...
        ELSE IF( NEWLOW.GT.OLDLOW )THEN
C          Need to send data in lower direction
           START=OLDLOW
           SECTION=NEWLOW-OLDLOW
           ...
        END IF

C       Obtain the old and new upper limit in the Migration Dimension (MD) for
C       the migrating processor
        OLDHIGH=OLDLIMIT(MD,2)
        NEWHIGH=NEWLIMIT(MD,2)
C       Examine the upper limits of migrating processor
        IF( NEWHIGH.GT.OLDHIGH )THEN
C          Need to receive data in upper direction
           START=OLDHIGH+1
           SECTION=NEWHIGH-OLDHIGH
           ...
        ELSE IF( NEWHIGH.LT.OLDHIGH )THEN
C          Need to send data in upper direction
           START=NEWHIGH+1
           SECTION=OLDHIGH-NEWHIGH
           ...
        END IF
```

Figure 3.58: Code used to determine the amount to migrate (SECTION) and from where to begin migrating (START) for the Migration Dimension

The new starting location (START_ADD) in memory from which to migrate the data can be found by offsetting the starting address (passed into the utility) by the value of START (evaluated in Figure 3.58). The internal communications operate in bytes, meaning that the starting address will be offset by a number of bytes, as demonstrated in Figure 3.59. The value of START_IND will always be less than or equal to the value of START, since it is passed into the migration utility as the low declared limit. Note that the staggered limits are not considered with this utility (CAP_MIGRATE), but will be involved with CAP_DLB_MIGRATE which is discussed in the next Section.

```
C       Evaluate the new starting address in memory from which to migrate the
C       data
        START_ADD=1+CAP_TYPELENS(ITYPE)*
     +                          (START_STRIDE*(START-START_IND))
```

Figure 3.59: Code used to determine the starting address of the internal communication (that operates in terms of bytes).

As stated earlier, the migrated data will be buffered into a contiguous section of memory using temporary storage, where the data is packed into a buffer before sending, and unpacked after receiving. The amount of continuous data (NITEMS) to communicate is evaluated as shown in Figure 3.60, observing the fact that the communication is operating in bytes.

```
C       Evaluate the size of the continuous migration message
        NITEMS=CAP_TYPELENS(ITYPE)*
     +                  NS1*NS2*NS3*NS4*NS5*NS6*SECTION
```

Figure 3.60: Code used to determine the amount of continuous data to communicate internally, which shall operate in bytes.

To carry out multi-buffering call to CAP_M_PACK is used before sending data (NEWLOW>OLDLOW, or NEWHIGH<OLDHIGH) as seen in Figure 3.61. The data (A) is buffered starting from the specified START_ADD into BUFF(*), where some of the parameters of the CAP_MIGRATE utility are passed into the call to CAP_M_PACK. Note that dummy parameters are used for the last set of S and NS (before NITEMS), since the CAP_M_PACK utility is used for both CAP_MIGRATE and CAP_DLB_MIGRATE (Section 3.7.6). Similarly, a call to CAP_M_UNPACK is used after receiving the buffered data (NEWLOW<OLDLOW, or NEWHIGH>OLDHIGH), also shown in Figure 3.61.

```
C       Temporarily pack the data (A) into a continuous section of
C       memory (BUFF)
        CALL CAP_M_PACK(A(START_ADD),S1,NS1,S2,NS2,
     +                  S3,NS3,S4,NS4,S5,NS5,S6,NS6,
     +                  SECTION,STRIDE,1,1,NITEMS,BUFF,ITYPE)
C       Send packed data


C       Receive packed data
C       Unpack the temporarily continuous section of memory (BUFF) into A
        CALL CAP_M_UNPACK(BUFF,A(START_ADD),S1,NS1,S2,NS2,
     +                    S3,NS3,S4,NS4,S5,NS5,S6,NS6,
     +                    SECTION,STRIDE,1,1,ITYPE)
```

Figure 3.61: Calls to pack and unpack continuous data into and from a buffer that are used inside the CAP_MIGRATE utility.

When CAP_MIGRATE is used to migrate data, the internal communication is with an immediate neighbour (since the staggered limits do not affect the migration of data in this Migration Dimension). The CAP_SEND and CAP_RECEIVE utilities (discussed in Section A.3.3) can therefore be used to migrate the continuous section of buffered data, as demonstrated in Figure 3.62. For example, having packed the data into BUFF(*), a continuous section (NITEMS) of type CAP_BYTES will be sent in the lower direction (PID=-((2*MD)-1)). If migrating in the first partitioned dimension (MD=1) then the data would be sent in the Left (-1) direction, whereas if MD=2 then the data would be sent in the Up (-3) direction. Similarly, a continuous section of buffered data will be received from the upper direction (PID=-(2*MD)), where the communication would be in the Right direction if MD=1, or in the Down direction if MD=2.

```
C       Pack data into BUFF
C       Send buffered data in the lower direction
        CALL CAP_SEND(BUFF,NITEMS,CAP_BYTES,-((2*MD)-1))


C       Receive buffered data from the upper direction
        CALL CAP_RECEIVE(BUFF,NITEMS,CAP_BYTES,-(2*MD))
C       Unpack data from BUFF
```

Figure 3.62: Communication calls that are used internally within the CAP_MIGRATE utility, where NITEMS of continuous data (in terms of bytes) are communicated in the specified communication direction starting from BUFF(*).

3.7.6 The Load Migration Algorithm For CAP_DLB_MIGRATE

When CAP_DLB_MIGRATE is used to migrate data, several internal communications may be required due to the effect of the staggered limits, unlike the single communication with an immediate neighbour used by CAP_MIGRATE (Section 3.7.5). The algorithm for this utility is not unlike that describing CAP_MIGRATE, since they differ only in relation to the Staggered Dimension (and its relevant parameters). Having determined that the data needs to be migrated in the Migration Dimension (a Non-Staggered Dimension), the next stage is to identify the neighbours to communicate with, as shown in Figure 3.63. Note that CAP_DLB_STAG_DIM is the number identifying the partition pass in which the Staggered Dimension was created (i.e. has a value of 3 for the example in Figure 3.51).

```
C       Examine each potential neighbour
        DO I=1,CAP_DNPROC(CAP_DLB_STAG_DIM)
C         Extract the I^th neighbour of migrating processor in the given direction
C         If examining the lower limits then DIRECTION=-((2*MD)-1)
C         If examining the upper limits then DIRECTION=-(2*MD)
          NEIGHBOUR=ALLNEIGHBOURS(I,DIRECTION)
          IF( NEIGHBOUR.NE.0 )THEN
C           Determine whether to communicate with this neighbour
            ...
          END IF
        END DO
```

Figure 3.63: Basic code that is used to identify neighbouring processors with which to communicate with.

As with finding the starting location and the amount to migrate in the Migration Dimension, the starting location in the Staggered Dimension (STAG_START) and the amount of data to migrate in the Staggered Dimension (STAG_SECTION) needs to be evaluated, as they differ for each internal communication. A comparison of the staggered limits of the migrating processor and its neighbour needs to be performed, as shown in Figure 3.64. A communication will only occur with a neighbour whose staggered limits overlap, where the staggered limits are extracted from CAP_DLB_PROCLIMITS (the old staggered limits are used since they will not have been updated, as the Staggered Dimension will always be processed last). Note the similarity with the DLB communications introduced earlier in Section 3.3.

```
C       Evaluate the beginning of the overlapping region with this neighbour
        SD1=(CAP_DLB_STAG_DIM*2-1)
        STAG_START=MAX(CAP_DLB_PROCLIMITS(SD1,CAP_PROCNUM),
     +                          CAP_DLB_PROCLIMITS(SD1,NEIGHBOUR))
C       Evaluate the end of the overlapping region with this neighbour
        SD2=(CAP_DLB_STAG_DIM*2)
        STAG_END=MIN(CAP_DLB_PROCLIMITS(SD2,CAP_PROCNUM),
     +                          CAP_DLB_PROCLIMITS(SD2,NEIGHBOUR))

C       Work out the message length (overlapping region)
        STAG_SECTION=STAG_END-STAG_START+1

C       Communicate with overlapping neighbour
        IF( STAG_SECTION.GT.0 )THEN
          ...
        END IF
```

Figure 3.64: Code used to determine STAG_START and STAG_SECTION, where a communication is performed with the neighbouring processor if its staggered limits overlap with the staggered limits of the migrating processor.

The new starting location in memory from which to migrate the data (START_ADD) can be found by offsetting the starting address (passed into the utility) by the value of START and also by STAG_START, as shown in Figure 3.65. The amount of continuous data to be buffered (NITEMS) must also take the alteration in the Staggered Dimension into consideration.

```
C       Evaluate the new starting address in memory from which to migrate the
C       data
        START_ADD=1+CAP_TYPELENS(ITYPE)*
     +                          ((START_STRIDE*(START-START_IND))+
     +                           (STAG_STRIDE*(STAG_START-STAG_IND)))

C       Evaluate the size of the continuous migration message
        NITEMS=CAP_TYPELENS(ITYPE)*NS1*NS2*NS3*NS4*NS5*NS6
     +                                  *SECTION*STAG_SECTION
```

Figure 3.65: Code used to determine the starting address of the internal communication, and the number of continuous bytes of data to be communicated.

The data needs to be packed into a buffer (BUFF) and sent, or received and unpacked from the buffer, in the same manner used for CAP_MIGRATE, as demonstrated in Figure 3.66. Notice that the only difference is that STAG_SECTION and STAG_STRIDE are now included, eliminating the need to use dummy parameters (set at 1 in Figure 3.61).

```
C       Temporarily pack the data (A) into a continuous section of
C       memory (BUFF)
        CALL CAP_M_PACK(A(START_ADD),S1,NS1,S2,NS2,
     +                  S3,NS3,S4,NS4,S5,NS5,S6,NS6,
     +                  SECTION,STRIDE,STAG_SECTION,
     +                  STAG_STRIDE,NITEMS,BUFF,ITYPE)
C       Send packed data

C       Receive packed data
C       Unpack the temporarily continuous section of memory (BUFF) into A
        CALL CAP_M_UNPACK(BUFF,A(START_ADD),S1,NS1,S2,NS2,
     +                    S3,NS3,S4,NS4,S5,NS5,S6,NS6,
     +                    SECTION,STRIDE,STAG_SECTION,
     +                    STAG_STRIDE,ITYPE)
```

Figure 3.66: Calls to pack and unpack continuous data into and from a buffer that are used inside the CAP_DLB_MIGRATE utility (which now involve STAG_SECTION and STAG_STRIDE).

A low-level communication with a specified neighbour is used to migrate the data in the Non-Staggered Dimension (illustrated in Figure 3.67), since it is no longer possible to communicate in a specified direction with a single neighbour, as the processor may potentially communicate with several neighbours.

```
C       Pack data into BUFF
C       Send buffered data to a specific neighbour using a low-level call
        CALL CAP_LOW_SEND(BUFF,NITEMS,CAP_BYTES,NEIGHBOUR)

C       Receive buffered data from a specific neighbour using a low-level call
        CALL CAP_LOW_RECEIVE(BUFF,NITEMS,CAP_BYTES,
     +                                              NEIGHBOUR)
C       Unpack data from BUFF
```

Figure 3.67: The low-level communication calls that are used internally within the CAP_DLB_MIGRATE utility, where NITEMS of continuous data (in terms of bytes) are communicated to a specific NEIGHBOUR starting from BUFF(1).

3.7.7 General Overview Of The Migration Utilities

Generic migration utilities are used to transfer data from one processor to another based on the newly calculated distribution. Table 3.12 gives a general overview of the process involved in constructing a communication call to migrate the data. For example, if the new lower limit is less than the old lower limit then the difference will be received from the processor 'below', starting from the new lower limit, and then unpacked. The data is communicated by specifying a communication

direction (if using CAP_MIGRATE), or by specifying a particular neighbour to communicate with (if using CAP_DLB_MIGRATE), where 'below' refers to Left/Up/Back, and 'above' refers to Right/Down/Forth, depending on the specified Migration Dimension.

NEW		OLD	PROCEDURE	DIRECTION	START
Low	<	Low	Receive and Unpack	From below	NewLow
Low	>	Low	Pack and Send	To below	OldLow
High	<	High	Pack and Send	To above	NewHigh+1
High	>	High	Receive and Unpack	From above	OldHigh+1

Table 3.12: Determination of required communication to set up new data partition.

A good reason for using generic migration calls is to hide the migration code from the user, as this can seem unnecessarily daunting. The migration calls calculate the amount of data to be migrated that ensure each processor owns the data defined by their new processor partition range limits, thus allowing the DLB strategy to work. Without this stage the whole DLB strategy could not be implemented correctly, as processor ownership would not be enforced.

3.8 Multi-Buffering

It may be necessary to communicate data in more than one buffered dimension when migrating from one processor to another, and so the following utilities have been developed to buffer the migrated data into temporary storage (a buffer) to avoid major changes to the user's code. Rather than have several communications looping through the different dimensions, it would be wiser to minimise the communication startup latency by minimising the number of communications needed to achieve the same result. The buffering is handled internally within the migration call, and so the user is unaware of the underlying operations, avoiding the need to clutter the user's code. Speed is essential and so multi-buffering can be used to avoid having to communicate segments of the migrated data rather than as much data as possible. A visible restriction with multi-buffering is the size of the

buffer, which is rectified by a buffering loop placed around the internal communication in the migration routines for when the buffer size is exceeded.

The parameters of the packing and unpacking utilities are similar to those of the migration calls themselves, except for the amount being migrated in the Migration Dimension and the Staggered Dimension, since these are calculated within the migration call and may change. CAP_M_PACK is used to take a variable, starting at a given location in memory, and starts packing the data into BUFF, where the value of COUNT has to be adjusted, as demonstrated in Figure 3.68. The same utility can be called from CAP_MIGRATE and CAP_DLB_MIGRATE, where STAG_SECTION and SATG_STRIDE are set to 1 (as dummy parameters) when called from CAP_MIGRATE.

Similarly, CAP_M_UNPACK is used to take the data out from BUFF, and starts placing it into the variable (A), starting at a given location in memory, as demonstrated in Figure 3.69. Once again the same utility can be called from CAP_MIGRATE and CAP_DLB_MIGRATE, avoiding the need to have a separate utility to deal with STAG_SECTION and STAG_STRIDE.

```
      SUBROUTINE CAP_M_PACK(A,SECTION,STRIDE,S1,NS1,S2,NS2,
     +                      S3,NS3,S4,NS4,S5,NS5,S6,NS6,
     +                      S7,NS7,ITYPE,BUFF,COUNT)
C     Declarations
C     Note: S1=STAG_SECTION and NS1=STAG_STRIDE when called from
C     CAP_DLB_MIGRATE; S7=1 and NS7=1 when called from
C     CAP_MIGRATE, as STAG_SECTION or STAG_STRIDE not used
      COUNT=1
      DO IS7=1,NS7
        DO IS6=1,NS6
          DO IS5=1,NS5
            DO IS4=1,NS4
              DO IS3=1,NS3
                DO IS2=1,NS2
                  DO IS1=1,NS1
                    DO IMIG=1,SECTION
                      DO IC=1,CAP_TYPELENS(ITYPE)
                        BUFF(COUNT)=
     +                    A(1+(IC-1)+CAP_TYPELENS(ITYPE)*(
     +                    (IMIG-1)*STRIDE+(IS1-1)*S1+(IS2-1)*S2+
     +                    (IS3-1)*S3+(IS4-1)*S4+(IS5-1)*S5+
     +                    (IS6-1)*S6+(IS7-1)*S7))
     +                  COUNT=COUNT+1
                      END DO
                    END DO
                  END DO
                END DO
              END DO
            END DO
          END DO
        END DO
      END DO
      COUNT=COUNT-1
```

Figure 3.68: Utility used to pack multi-dimensional data into a buffer, which is called from within a migration call (CAP_MIGRATE or CAP_DLB_MIGRATE).

```
      SUBROUTINE CAP_M_UNPACK(BUFF,A,SECTION,STRIDE,S1,NS1,
     +          S2,NS2,S3,NS3,S4,NS4,S5,NS5,S6,NS6,S7,NS7,ITYPE)
C     Declarations
C     Note: S1=STAG_SECTION and NS1=STAG_STRIDE when called from
C     CAP_DLB_MIGRATE; S7=1 and NS7=1 when called from
C     CAP_MIGRATE, as STAG_SECTION or STAG_STRIDE not used
      COUNT=1
      DO IS7=1,NS7
        DO IS6=1,NS6
          DO IS5=1,NS5
            DO IS4=1,NS4
              DO IS3=1,NS3
                DO IS2=1,NS2
                  DO IS1=1,NS1
                    DO IMIG=1,SECTION
                      DO IC=1,CAP_TYPELENS(ITYPE)
                        A( 1+(IC-1)+CAP_TYPELENS(ITYPE)*(
     +                  (IMIG-1)*STRIDE+(IS1-1)*S1+(IS2-1)*S2+
     +                  (IS3-1)*S3+(IS4-1)*S4+(IS5-1)*S5+(IS6-1)*S6+
     +                  (IS7-1)*S7) )=BUFF(COUNT)
                        COUNT=COUNT+1
                      END DO
                    END DO
                  END DO
                END DO
              END DO
            END DO
          END DO
        END DO
      END DO
```

Figure 3.69: Utility used to unpack multi-dimensional data from a buffer, which is called from within a migration call (CAP_MIGRATE or CAP_DLB_MIGRATE).

3.9 Updating The Processor Partition Range Limits

The processor partition range limits have to be updated both in the user's code itself, and internally (for use in the DLB utilities). The processor partition range limits of a particular dimension need to be updated after migrating in that dimension, since the new limits are needed when migrating data in subsequent dimensions, as well as being used in the ensuing execution. Since the limits of each partitioned dimension need to be updated with their newly calculated values, a single utility can be called several times to carry this out.

The processor partition range limits being updated are passed into the call CAP_DLB_REASSIGNLOWHIGH along with the Migration Dimension, as demonstrated in Figure 3.70, where each of these parameters will differ whenever

this utility is called. The actual utility is shown in Figure 3.71, where the processor partition range limits are updated on every processor. Only one call is executed for potentially 100's of migrated variables in each dimension, keeping the code neat and simple.

```
C     Migrate data in the 1st Non-Staggered Dimension
      CALL CAP_DLB_REASSIGNLOWHIGH(CAP_LOW,CAP_HIGH,1)
      ......
C     Migrate data in the 2nd Non-Staggered Dimension
      CALL CAP_DLB_REASSIGNLOWHIGH(CAP2_LOW,CAP2_HIGH,2)
      ......
C     Migrate data in the Staggered Dimension
      CALL CAP_DLB_REASSIGNLOWHIGH(CAP3_LOW,CAP3_HIGH,3)
      ......
```

Figure 3.70: Code demonstrating how the processor partition range limits are updated after migration.

```
      SUBROUTINE CAP_DLB_REASSIGNLOWHIGH(LOW,HIGH,MD)
C     Declarations

C     Reassign the lower and upper limit of this processor in the given
C     dimension
      LOW=NEWLIMIT(MD,1)
      HIGH=NEWLIMIT(MD,2)

      END
```

Figure 3.71: The utility used to update the processor partition range limits after migration, where the limits and the Migration Dimension have been specified.

After load migration, before updating the halo region, the processor partition range limits need to be updated internally, which only needs to be done once. This essentially involves reassigning the CAP_DLB_PROCLIMITS to the corresponding values in CAP_DLB_NEW_PROCLIMITS (which were obtained when calculating the new limits, seen in Section 3.5). If the processor partition range limits were not updated, then the old processor partition range limits would be used in calls to any subsequent DLB utilities, such as the DLB communications for instance.

The utility call CAP_DLB_NEW2OLD_LIMITS, shown in Figure 3.72, has no parameters since all of the variables are stored internally. The values of the new processor partition range limits assigned on Processor 1 are broadcast out to the other processors.

```
      SUBROUTINE CAP_DLB_NEW2OLD_LIMITS()
C     Declarations

C     Only reassign limits if the main processor
      IF( CAP_PROCNUM.EQ.1 )THEN
C       Loop over the number of processors
        DO I=1,CAP_NPROC
C         Loop over the both limits of each dimension
          DO D=1,CAP_PROCDIM*2
            CAP_DLB_PROCLIMITS(D,I)=
     +                    CAP_DLB_NEW_PROCLIMITS(D,I)
          END DO
        END DO
      END IF

C     Broadcast newly assigned limits to all processors
      CALL CAP_MBROADCAST(CAP_DLB_PROCLIMITS(1,1),
     +                    CAP_MAXPROCDIM*CAP_PROCDIM*
     +                    CAP_NPROC,1)

      END
```

Figure 3.72: Utility used to update the internal processor limits.

3.10 Overview Of The DLB Utilities

A brief description of each of the main DLB utilities is given in Table 3.13, where their usage is demonstrated in the next Chapter.

CAP_DLB_SETALLNEIGHBOURS	- determines all possible neighbours (and cyclic neighbours) for each processor
CAP_DLB_SETUPLIMITS	- sets up the internal processor partition range limits (CAP_DLB_PROCLIMITS) for all processors
CAP_DLB_START_TIMER	- starts timing the imbalanced loop
CAP_DLB_STOP_TIMER	- stops timing the imbalanced loop (called by CAP_DLB_DECIDE)
CAP_DLB_DECIDE	- stops timing the imbalanced loop (with a call to CAP_DLB_STOP_TIMER) and determines whether or not to calculate the new partition
CAP_DLB_START_REBAL	- starts timing the redistribution process that involves a call the routine to calculate the new processor partition range limits and a call to evaluate the risk of load migration
CAP_DLB_FINDNEWLIMITS	- calculates the new processor partition range limits (i.e. the new partition)
CAP_DLB_MIGRATE_RISK	- determines if the new partition should be implemented (i.e. is it worth migrating?)
CAP_DLB_STOP_REBAL	- stops timing the redistribution process, ands records the iteration in which the load was redistributed
CAP_DLB_REASSIGNLOWHIGH	- re-assigns the processor partition range limits on each processor to the newly calculated limits
CAP_DLB_NEW2OLD_LIMITS	- updates the internal processor partition range limits for use within the DLB utilities
DLB Communications utilities	- enables processors to communicate across staggered limits (to their non-immediate neighbour)
DLB Migration utilities	- transfers data between processors to satisfy the new partition, ensuring that each processor can correctly operate on the data that they now own
DLB Packing and Unpacking utilities	- used when migrating data, these utilities enable the transfer of multi-dimensional arrays between processors in one communication message

Table 3.13: Some of the devised DLB utilities with a brief explanation.

3.11 Summary

This Chapter discussed the set of generic utilities that are required to successfully implement the selected DLB strategy within a CAPTools generated parallel code. The main benefits of using generic utilities are that they attempt to minimise the changes to the user's code by hiding the underlying operations (which can be changed without affecting this code), and that they can be applied to a wide range of application codes. Utilities were needed to execute the parallel code in DLB mode, where the obvious differences when running in DLB mode compared to non-DLB mode had to be overcome. With the selected strategy each processor could no longer communicate with its immediate neighbour, and so a utility was used to set up a new communication topology identifying processors to communicate with, where the processor partition range limits of these neighbouring processors also had to be known.

The selected DLB strategy uses a mixture of coincidental and non-coincidental (staggered) limits, where a processor will only need to communicate with its immediate neighbour in the Staggered Dimension but may have several potential processors to communicate with in a Non-Staggered Dimension. Some DLB communication calls were therefore devised that enable processors to communicate across the staggered limits, where these calls, like existing CAPTools communications, can deal with any data type by operating in bytes.

Having successfully tested the DLB communications it was possible to develop the remaining utilities, which dealt with the actual dynamic load balancing. A utility was devised that decided when to redistribute the load, which based the decision on the processor timings of a particular iteration of an imbalanced loop. If the load did need to be redistributed, then it was necessary to have a utility that calculated the new processor workload (defined by the processor partition range limits). The new workload was calculated separately for each partitioned dimension, from which the new limits could be evaluated. The importance of the problem's imbalance classification has been realised, where a problem could contain either processor imbalance or physical imbalance. The algorithm to calculate the new load had to consider both processor and physical imbalance in order to be generic, as this factor affected the new distribution. The

algorithm basically reduced the workload on the slow or heavily loaded processors, and placed it onto the faster or lighter processors. Cells were lost at a processor's own weight (time to process a cell), but were gained at either the processor's own weight (with processor imbalance), or at a neighbouring weight (with physical imbalance). To account for the balance already obtained whilst processing the current dimension, it was necessary to adjust the processor timings before proceeding to balance subsequent dimensions.

It was decided that the partition would remain the same if the amount of data to move was insufficient, since it would not be worthwhile to migrate the data. A utility was therefore used to validate the new distribution, after which the new distribution could be implemented.

The processor ownership of data had to be ensured in order to implement the new distribution, such that each processor owned the newly defined data that it would subsequently operate upon. Data needed to be transferred between neighbouring processors onto the new owners, which could only be done using communication calls to physically move the data. Generic migration utilities were therefore developed to move data from one processor to another without cluttering the user's code unnecessarily with several communication calls for each migrated variable. One migration utility would be used to migrate data in the Non-Staggered Dimensions, and another would be used to migrate data in the Staggered Dimension, where each dimension would be processed separately using the old processor partition range limits of those dimensions not yet migrated. The underlying operation of the migration utilities is similar in nature to the devised DLB communication, where an internal communication is used to migrate a calculated amount of data (based on the comparison of the old and new limits) with a neighbouring processor. The neighbour is explicitly identified when migrating in a Non-Staggered Dimension, but is specified by a communication direction when migrating in the Staggered Dimension.

The processor partition range limits for a particular dimension, which are used within the user's code, need to be updated after migrating the data in that dimension, as these will be used when migrating data in subsequent dimensions. The internal processor partition range limits (used in DLB communications for instance) also need to be updated, which can be done after migrating the data in all of the partitioned dimensions. All that remains to implement the new partition

is the duplication of overlap (halo) communications, which is covered in Section 4.7.3 of the next Chapter. Therefore the implementation of the selected DLB strategy is now possible using these utilities, which can be implemented manually as discussed in the next Chapter.

Chapter 4 Manually Implementing The DLB Staggered Limit Strategy Within A CAPTools Generated Parallel Structured Mesh Code

The previous two Chapters discussed the DLB Staggered Limit Strategy and the utilities needed to implement this approach within an application code. This Chapter deals with the issue of manual implementation, examining how to improve the parallel performance of a code using the DLB Staggered Limit Strategy and its generic utilities and finally devising details of a generic approach for real codes.

There are several stages involved in manually implementing DLB within a parallel code, This DLB strategy shall contain non-coincidental processor partition range limits, so those communications generated in previously partitioned dimensions (in the Non-Staggered Dimensions) will need to be converted into DLB communications. Due care is needed if the whole DLB code is to operate correctly, as failing to convert just one communication will introduce errors. As well as converting existing communications into DLB communications the main load balancing code needs to be inserted, which will obtain the new data distribution and ensure processor ownership of data. The following Sections will demonstrate how tedious and time consuming the process of manually implementing this DLB strategy will be, and why much time and effort can be saved simply by automating the whole process.

4.1 The Implementation Algorithm

The new code that is to be inserted into the original parallel code should be understandable and unobtrusive to the user, such that the user can maintain their code without needing to know the underlying operations of the inserted DLB code in detail. The main algorithm steps that are used to dynamically load balance a parallel code are shown in Figure 4.1.

- *Change existing communication calls of previously partitioned data (in the Non-Staggered Dimensions) into the new DLB communication calls*
- *Insert dynamic load balancing code:*
    - *Set up parallel code to execute in DLB mode*
    - *Start/Stop timer*
    - *Calculate new processor partition limits*
    - *Add migration calls*
    - *Assign new limits*
    - *Duplicate overlap communications*

Figure 4.1: The basic DLB algorithm used to implement the DLB Staggered Limit Strategy within a parallel code.

4.2 Setting Up The DLB Parallel Code

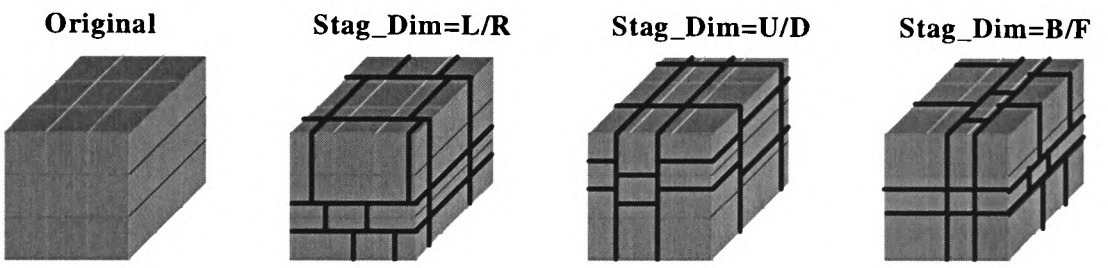
The Staggered and Non-Staggered Dimensions need to be identified before converting a CAPTools generated parallel code into a DLB parallel code, as it would be impossible to implement the algorithm in Figure 4.1 without knowing which dimension contained the staggered limits. The parallel code needs to be set up to execute in DLB mode, therefore the DLB parameters need to be initialised.

4.2.1 The Staggered And Non-Staggered Dimensions

The Non-Staggered Dimensions contain coincidental processor partition range limits, whereas the Staggered Dimension contains non-coincidental processor partition range limits. During the manual implementation of the selected DLB strategy, both the Staggered and Non-Staggered Dimensions will be referred to many times and so these should be defined. For reasons that shall be made clearer in Chapter 5, it has been decided that the Staggered Dimension (CAP_DLB_STAG_DIM) shall be the last partitioned dimension, although in this Section it would be possible for the user to simply select any partitioned dimension to be the Staggered Dimension because all of the communications have already been generated (Section 5.3). The user can determine all of the necessary information needed to convert an existing parallel code into a DLB parallel code

and so it makes no difference which dimension is chosen to contain the staggered limits. Using a 2D problem for example, CAPTools would have generated the Left/Right processor partition range limits before generating the Up/Down limits, which would mean that the Staggered Dimension would contain the limits of the second partition, as shown in Figure 2.10, i.e. CAP_DLB_STAG_DIM=2. If a 3D problem were used, then the Staggered Dimension would contain the Back/Forth limits since these were generated last, implying that Non-Staggered Dimensions would contain the Left/Right and Up/Down limits, as seen in Figure 4.2 and CAP_DLB_STAG_DIM=3.

Note that each dimension can be referred to either through an index, dimension, or stride. It is important to know exactly which dimension contains the staggered limits, as this also reveals which dimensions contain the non-staggered limits. Less effort is needed to generate a DLB parallel code if this fact is known, as there would be no need to convert the communications in all dimensions into DLB communications.



Find new:	Row widths	Col widths	Col widths
Shift workload:	Up/Down	Left/Right	Left/Right
Find new:	Plane widths	Plane widths	Row widths
Shift workload:	Back/Forth	Back/Forth	Up/Down
Find new:	Individual processor widths in the Staggered Dimension and then shift workload		

Figure 4.2: Illustration of a 3D problem in which different dimensions have been staggered.

4.2.2 Initialising DLB Mode

As with CAP_INIT (Section A.2), which sets up an application code to run in parallel, this parallel code needs to run in DLB mode, which means calling the set-up utilities inside the main program before any DLB code is used (such as a DLB communication for instance). This can be achieved by inserting a call to

CAP_DLB_SETUPALLNEIGHBOURS (Section 3.2.1) after the declarations in the main program (after CAP_INIT). Additionally, a call to CAP_DLB_SETUPLIMITS (Section 3.2.2) should be inserted after setting up the processor partition range limits in the parallel code. Any DLB variables that are needed within the DLB Routine for load balancing and for load migration need to be declared and initialised after the existing declarations within the DLB Routine, as shown in Figure 4.3.

```
      PROGRAM MAIN
C     Declarations
C     DLB Declarations
C     DLB Initialisations
      CALL CAP_INIT
      CALL CAP_DLB_SETUPALLNEIGHBOURS
      CALL CAP_SETUPDPART(1,NI,CAP1_LOW,CAP1_HIGH,1)
      CALL CAP_DLB_SETUPLIMITS(CAP1_LOW,CAP1_HIGH,1)
      CALL CAP_SETUPDPART(1,NJ,CAP2_LOW,CAP2_HIGH,2)
      CALL CAP_DLB_SETUPLIMITS(CAP2_LOW,CAP2_HIGH,2)
```

Figure 4.3: Setting up code to run in DLB mode.

4.3 Converting Existing Communications Into DLB Communications

At present, communications usually occur between immediately neighbouring processors who share the same processor partition range limits, which means that a processor will rarely have to communicate with a non-neighbouring processor. However, the selected DLB strategy uses non-coincidental limits in one of the partitioned dimensions, meaning that the communication topology is now different to that of the non-DLB parallel communication topology so that existing communications should either be replaced by new communications or they should be altered to incorporate this change. A fair amount of work would be involved if the existing communications were to be replaced completely, as the relevant communications would have to be identified and then deleted, and then finally the new communications would be inserted. Therefore in an attempt to minimise the changes to the user's code, it is better to simply alter existing communications

such that the same underlying operation is performed, allowing the user to maintain and optimise their code easily.

In order to implement the DLB strategy each processor must be able to communicate with several processors in any of the Non-Staggered Dimensions, whereas it is still only necessary to communicate with immediately neighbouring processors in the Staggered Dimension. The user must therefore reconfigure the communication topology to allow processors to also communicate with processors other than their immediate neighbours (Section 4.2).

The communications that need to be converted therefore first need to be identified, which essentially means examining all of the communications generated in a Non-Staggered Dimension, as only these communications may need to be changed.

Once a communication has been identified as being a potential candidate for conversion into a DLB communication, it is then necessary to determine whether or not the communication needs to be changed. Conversion into a DLB communication is only necessary if the communicated data is affected by the staggered limits, which means detecting if the data is also partitioned in the Staggered Dimension. This means that the user must search for statements involving the communicated data and the processor partition range limits of the Staggered Dimension, or seeing whether the communication itself contains the staggered limits, as demonstrated in Figure 4.4.

```
DO I=MIN(1,CAP1_LOW),MAX(NI-1,CAP1_HIGH-1)
   DO J=MIN(1,CAP2_LOW),MAX(NJ,CAP2_HIGH)
      A(J,I)=U(J,I+1)*PI
   END DO
END DO                              1) usage of communicated data that is partitioned in
                                       the Staggered Dimension

...

CALL CAP_RECEIVE(U(CAP2_LOW,CAP1_HIGH+1),(CAP2_HIGH-CAP2_LOW+1),
                                 2,CAP_RIGHT)

                                    2) communication involves staggered partition
```

Figure 4.4: The communication involving U can be converted into a DLB communication as 1) there exists a statement involving the use of the partitioned limits and the communicated data; and 2) the communication itself involves the staggered processor partition range limits.

The appropriate values of FIRST, STAG_STRIDE, LOWLIM and HIGHLIM are entered into the parameter list of the call (Section 3.3). To start with, the communication call name is altered to indicate that this is now a DLB communication that will allow processors to communicate across staggered limits, as seen in Figure 4.5. The next stage is to extract the relevant information from the communication call and from the appropriate statements to complete the parameter list of the new call.

Information relating to the communicated data:
real U(100,200) ← in declaration statement

Pass	Index	Stride	Processor Partition Range Limits
1	2	100	CAP1_LOW/CAP1_HIGH
2	1	1	CAP2_LOW/CAP2_HIGH

Original communication call:

```
CALL CAP_RECEIVE        (U(CAP2_LOW,CAP1_HIGH+1),
                         U(CAP2_LOW,CAP1_LOW),
                         (CAP2_HIGH-CAP2_LOW+1),
                         2,CAP_RIGHT)
```

New DLB communication call:

```
CALL CAP_DLB_RECEIVE    (U(CAP2_LOW,CAP1_HIGH+1),
                         U(CAP2_LOW,CAP1_LOW),
                         (CAP2_HIGH-CAP2_LOW+1),
                         CAP2_LOW,1,CAP2_LOW,CAP2_HIGH,
                         2,CAP_RIGHT)
```

change communication name

add in FIRST, the staggered component extracted from the starting address

add in the STAG_STRIDE (in this case 1)

add in LOWLIM and HIGHLIM, the staggered partition range limits

Figure 4.5: Transformation of a communication into a DLB communication (along with information relating to the communicated data).

In order to obtain the starting position of the communicated data in the Staggered Dimension (FIRST), the user must know which dimension of the data contains the staggered limits. In the example shown in Figure 4.6, the third dimension was partitioned with the knowledge that its limits may be staggered,

and so the third index of the starting address is extracted. It would be incorrect to simply use the low declared limit in the Staggered Dimension, as the communicated data may not start from this position (it will typically start from cap_low), which is why FIRST must be extracted from the starting address stated in the call. If the given starting address is 1D mapped then the user must determine the value of FIRST using the declaration statement of the communicated data (Figure 4.6).

real U(20,10,30,40) ← in declaration statement

1st partitioned index=1 => stride=1
2nd partitioned index=**3** => stride=**200** (Staggered Stride)

Starting Address:	FIRST:	Explanation of how to obtain FIRST:
U(cap1_high+1,1,**cap2_low**,d)	cap2_low	third component in address (partitioned index in the Staggered Dimension
U(1+1*((cap1_high+1)-1) +20*(1-1)+200*(**cap2_low**-1)+6000*(d-1))	cap2_low	this component relates to the staggered stride (200), and because this term contains the processor partition range limit of the Staggered Dimension
U(1+1*((cap1_high+1)-1) +20*(1-1)+200*(**34**-1) +6000*(d-1))	34	this component relates to the staggered stride (200)

Figure 4.6: How to obtain FIRST when multi-dimensional arrays or 1D mapped indices are used.

Knowing the Staggered Dimension and the declaration statement of the communicated data STAG_STRIDE would be set to 200 (Figure 4.6). Although the staggered stride may be found within the communication call, this will not always be the case, which is why it should be extracted from the declaration statement.

The remaining two parameters, LOWLIM and HIGHLIM, are used to find the offsets to the processor partition range limits, which are usually going to be set to the staggered limits since most communications tend not to extend beyond these limits. The user must determine whether or not the communication only involves data between the staggered processor partition range limits. This can be done by examining the communication and its surrounding statements as demonstrated in Figure 4.7. If the communication is contained within a DO Loop

for instance, then it may be that the communicated data in the Staggered Dimension is related to the loop limits. The communication has already been partitioned in the Staggered Dimension during the parallelisation process (Section A.3.3) and so the user could simply examine the communication call to calculate whether the communication only involves data between the processor partition range limits.

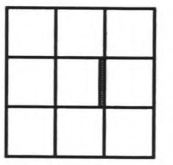
real U(200,300) ← in declaration statement

1st partitioned index=1
2nd partitioned index=2 (Staggered Dimension)

```
DO CAP_J=CAP2_LOW,CAP2_HIGH
   CALL CAP_SEND(U(CAP1_HIGH,CAP_J),1,2,CAP_RIGHT)
END DO
```

is the same as:

```
CALL CAP_BSEND(U(CAP1_HIGH,CAP2_LOW),1,200,
                        (CAP2_HIGH-CAP2_LOW+1),2,
                        CAP_RIGHT)
```



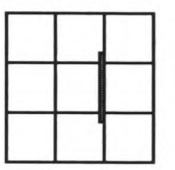
There are no 'offsets' – each processor will communicate between their processor partition range limits.

LOWLIM=CAP2_LOW, HIGHLIM=CAP2_HIGH

```
DO CAP_J=MAX(1,CAP2_LOW-2),MIN(NJ,CAP2_HIGH+3)
   CALL CAP_SEND(U(CAP1_HIGH,CAP_J),1,2,CAP_RIGHT)
END DO
```

is the same as:

```
CALL CAP_BSEND(U(CAP1_HIGH,MAX(1,CAP2_LOW-2)),1,
                        200,(MIN(NJ,CAP2_HIGH+3)-MAX(1,
                        CAP2_LOW-2)+1),2,CAP_RIGHT)
```



There are some 'offsets' (-2 and +3 in the lower and upper directions respectively) – this information needs to be known internally by the DLB communication utilities.

LOWLIM=CAP2_LOW-2, HIGHLIM=CAP2_HIGH+3

Figure 4.7: Determination of LOWLIM and HIGHLIM.

There are certain situations which require more care when converting communications into DLB communications, such as communications that are masked in the Staggered Dimension or when the data appears to be unpartitioned. In Figure 4.8 for example, if the communication is inside an execution control mask then the communication only occurs between processors who own the

specified value in the mask, and so FIRST is set to be the mask value (3) and STAG_STRIDE is set to zero (Section 3.3.4). The remaining two parameters are not used within the underlying operation and so they are set to the processor partition range limits.

```
DO J=1,NJ
   IF( 3.LE.CAP2_HIGH .AND. 3.GE.CAP2_LOW )THEN
      DO I=MAX(1,CAP1_LOW),MIN(NI,CAP1_HIGH)
         U(I,J)=...
      END DO
   END IF
END DO
...
IF(3.LE.CAP2_HIGH .AND. 3.GE.CAP2_LOW )THEN
   CALL CAP_BSEND(U(CAP1_LOW,1),1,200,NJ,2,CAP_RIGHT)
END IF
```

FIRST=3, STAG_STRIDE=0, LOWLIM=CAP2_LOW, HIGHLIM=CAP2_HIGH

Figure 4.8: Constructing a 'special' DLB communication, in which only specific processors will be involved in the internal communications.

In the second situation, the user may come across a communication in which the data is said to be unpartitioned, implying for example that the same array has been used in different ways (Figure B.34). The user must be careful not to get confused between the various partitions as this could lead to an error being made. The user must determine if the communicated data is implicitly partitioned in the Staggered Dimension, which means examining all of the assignment statements of the data to see whether they are masked in the Staggered Dimension. If all of the assignment statements are masked in the Staggered Dimension then the communication should be converted into a DLB communication, otherwise if any one of the assignment masks differ then the data should not be treated as if it were partitioned (Section B.9.1.2).

Consider for example the routine PINTGR in the APPLU_1.4 code (an extract of which is shown in Figure 4.9), where the variable U has first been partitioned in its fourth dimension and then partitioned in its third dimension (i.e. the Staggered Dimension involves the third index, J). The variables PHI1 and PHI2 are implicitly partitioned, as they do not retain the same partitioning throughout the entire code. When involved in the calculation of FRC1, these variables are implicitly partitioned in index 2 based on the J component of U.

When involved in the calculation of FRC2 they are again implicitly partitioned in index 2, but in this instance the partition is based on the K component of U. Note that in this example, the transformations needed to convert the necessary communications into DLB communications are shown in bold and are also highlighted, whilst any code information used to make a transformation is just bold.

Examining the first communication (the CAP_BSEND involving PHI2), the initial stage is to identify that this communication needs to be converted. The assignment statement of this communicated data is the first assignment of PHI2, which involves the use of U that is partitioned in the Staggered Dimension. If there were any other assignment statements relating to this communicated data then these would also need to be examined. It is possible to convert this communication into a DLB communication since there is a linear relationship involving J between the second index of PHI2 and the third index of U. The communication name is changed and FIRST is set to the second index of the communication's starting address (MAX(2,CAP2_LA)). The STAG_STRIDE is set to the stride of the second index of PHI2 (which is 12), and finally the LOWLIM and HIGHLIM must be set according to the mask in the Staggered Dimension. In this case, the second components of the lower and upper limits of the J loop (i.e. JI1+CAP2_LA-2 and JI1+CAP2_HA-1 respectively).

When examining the CAP_EXCHANGE of PHI2, it is seen (from the assignment mask) that the assignment only occurs on the processor owning JI2 in the Staggered Dimension. This suggests that a 'special' DLB communication is needed, with STAG_STRIDE set to 0 and LOWLIM and HIGHLIM simply set to CAP2_LA and CAP2_HA respectively. FIRST is then set as the constant component of the staggered mask (i.e. JI2 in this case). A similar process is used to determine the transformation of the second CAP_EXCHANGE, this time involving PHI1 where FIRST is set to 2, meaning that only those processors owning row 2 will need to be involved in the DLB communication.

```
SUBROUTINE PINTGR(CAP1_LA,CAP1_HA,CAP2_LA,CAP2_HA)
PARAMETER (ISIZ1=12,ISIZ2=12,ISIZ3=12)
COMMON /CVAR/U(5,ISIZ1,ISIZ2,ISIZ3),...
DIMENSION PHI1(ISIZ1,ISIZ2),PHI2(ISIZ1,ISIZ2)
...
READ*,NX,NY,NZ
II1=2
II2=NX-1
JI1=2
JI2=NY-2
KI1=3
KI2=NZ-1
...
IF (((KI2.LE.CAP1_HA).AND.(KI2.GE.CAP1_LA)).OR.
    ((3.LE.CAP1_HA).AND.(3.GE.CAP1_LA))) THEN
   DO J=MAX(JI1,JI1+CAP2_LA-2),MIN(JI2,JI1+CAP2_HA-1),1
      DO I=II1,II2,1
         IF ((3.LE.CAP1_HA).AND.(3.GE.CAP1_LA)) THEN
            PHI1(I,J)=...U(2,I,J,KI1)...
         END IF
         IF ((KI2.LE.CAP1_HA).AND.(KI2.GE.CAP1_LA)) THEN
            PHI2(I,J)=...U(2,I,J,KI2)...
         END IF
      END DO
   END DO
   IF ((KI2.LE.CAP1_HA).AND.(KI2.GE.CAP1_LA)) THEN
      CALL CAP_DLB_BSEND(PHI2(2,MAX(2,CAP2_LA)),II2-1,12,
                MIN((2+JI2-1)-1,CAP2_HA+1)-MAX(2,CAP2_LA)+1,
                MAX(2,CAP2_LA),12,CAP2_LA,CAP2_HA+1,42,CAP_LEFT)
   END IF
END IF
...
Other Left/Right communications involving PHI1 and PHI2
...
FRC1=0.0D+00
DO J=MAX(JI1,JI1+CAP2_LA-2),MIN(JI2-1,JI1+CAP2_HA-2),1
   DO I=II1,II2,1
      FRC1=FRC1+PHI1(I,J)+PHI1(I+1,J)+PHI1(I,J+1)+PHI1(I+1,J+1)+
           PHI2(I,J)+PHI2(I+1,J)+PHI2(I,J+1)+PHI2(I+1,J+1)
   END DO
END DO
CALL CAP_DCOMMUTATIVE(FRC1,42,CAP_R8ADD,CAP_UP2)
FRC1=DXI*DETA*FRC1


IF (((JI2.LE.CAP2_HA).AND.(JI2.GE.CAP2_LA)).OR.
    ((2.LE.CAP2_HA).AND.(2.GE.CAP2_LA))) THEN
   DO K=MAX(KI1,KI1+CAP1_LA-3),MIN(KI2,KI1+CAP1_HA-3),1
      DO I=II1,II2,1
         IF ((2.LE.CAP2_HA).AND.(2.GE.CAP2_LA)) THEN
            PHI1(I,K)=...U(2,I,JI1,K)...
         END IF
         IF ((JI2.LE.CAP2_HA).AND.(JI2.GE.CAP2_LA)) THEN
            PHI2(I,K)=...U(2,I,JI2,K)...
         END IF
      END DO
   END DO
END IF
IF (((JI2.LE.CAP2_HA).AND.(JI2.GE.CAP2_LA)).AND.(2.LT.CAP2_LA)) THEN
   CALL CAP_BSEND(PHI2(2,MAX(1,CAP1_LA-2)+2),II2-1,12,
```

```
            MIN(KI2-3,CAP1_LA-2)-MAX(1,CAP1_LA-2)+1,42,CAP_UP2)
END IF
...
CALL CAP_DLB_EXCHANGE(PHI2(2,CAP1_LA+1),PHI2(2,CAP1_LA),II2-1,
       JI2,0,CAP2_LA,CAP2_HA,42,CAP_RIGHT)
...
CALL CAP_DLB_EXCHANGE(PHI1(2,CAP1_LA+1),PHI1(2,CAP1_LA),II2-1,
       2,0,CAP2_LA,CAP2_HA,42,CAP_RIGHT)

IF ((2.LE.CAP2_HA).AND.2.GE.CAP2_LA)) THEN
  FRC2=0.0D+00
  DO K=MAX(KI1,KI1+CAP1_LA-3),MIN(KI2-1,KI1+CAP1_HA-3),1
    DO I=II1,II2,1
      FRC2=FRC2+PHI1(I,K)+PHI1(I+1,K)+PHI1(I,K+1)+PHI1(I+1,K+1)+
           PHI2(I,K)+PHI2(I+1,K)+PHI2(I,K+1)+PHI2(I+1,K+1)
    END DO
  END DO
  CALL CAP_SEND(FRC2,1,42,CAP_DOWN2)
END IF
...
CALL CAP_DCOMMUTATIVE(FRC2,42,CAP_R8ADD,CAP_LEFT)
FRC2=DXI*DZETA*FRC2
```

Figure 4.9: Example from APPLU_1.4 in which some of the communications of the implicitly partitioned variables PHI1 and PHI2 have been converted into DLB communications.

This stage is important to the whole process of implementing the DLB Staggered Limit Strategy within a parallel code, as without this stage it would be impossible for processors to communicate with non-neighbouring processors. There may be hundreds of communications that need to be converted, each of which needs the care and attention of the user, as it is imperative to maintain the underlying operation of the communications whilst minimising the changes to the user's code. Many of the communications in the Non-Staggered Dimensions will therefore need to be converted in order to correctly handle the inter-processor communication across staggered limits. This task is greatly aided by the use of the browser windows in CAPTools that display all of the necessary information. Without the use of CAPTools, the implementation of the DLB Staggered Limit Strategy would be far more difficult and prone to many more errors.

4.4 Where To Redistribute The Workload

Given that the application code shall contain a DLB strategy, and having already initialised the code to execute in DLB mode with the chosen Staggered Dimension (Sections 4.2 and 4.3), the next stage is to determine where to redistribute the workload. Using a profiler or user knowledge, the location containing the load imbalance can be identified. If the application code is very large then it could be difficult for the user to identify the exact location at which to redistribute the workload.

Load imbalance will usually be in called routines, and so the user must select a loop containing calls to as many as possible of these routines, as this is where most of the efficiency can be improved (Section 2.7.1). It is important to place the DLB code in the correct location within an application code (i.e. around key sections of the code in the selected loop) otherwise the load imbalance may not be identified and dealt with, defeating the purpose of DLB. The timers are placed around code that is executed by each processor. A 'loop' is the ideal place in which to place the DLB code as each processor is executing the same block of code for a given number of iterations (Figure 4.10), where it is inside this DLB Loop that the load imbalance can become significant and the idle time inside this loop can dominate the runtime of the whole application code. An informed decision at this stage is necessary if the parallel performance is to be improved, as the performance gain may not be significantly noticeable if DLB is performed at the wrong location.

```
        DO I=1,NI
           ...
           CALL SOLVE()
           ...
        END DO
```

```
        ...
        I=1
10      CONTINUE
        I=I+1
        ...
        CALL SOLVE()
        ...
        IF(DIFF.GT.TOL)THEN
           GOTO 10
        END IF
```

```
        ...
        I=1
10      CONTINUE
        I=I+1
        ...
        IF(...)THEN
           GOTO 10
        ELSE
           GOTO 20
        END IF
        ...
20      CONTINUE
        ...
        IF(...)THEN
           GOTO 10
        END IF
```

Figure 4.10: Possible DLB Loops, where most of the processing is performed inside the loop.

The imbalance within the DLB Loop is determined by placing some timers within the DLB Loop and ascertaining whether or not the load imbalance is significant. A start and stop timer should be placed around the iteration block, where the user must decide in which order to place them. Timers are placed around this code, where the load can be redistributed either at the beginning of the loop iteration, or at the end, as shown in Figure 4.11.

```
DO I=1,NI
   REDISTRIBUTE?
   START TIMER
   ...
   CALL SOLVE()
   ...
   STOP TIMER
END DO
```

```
DO I=1,NI
   START TIMER
   ...
   CALL SOLVE()
   ...
   STOP TIMER
   REDISTRIBUTE?
END DO
```

```
      I=1
10    CONTINUE
      REDISTRIBUTE?
      START TIMER
      ...
      CALL SOLVE()
      ...
      IF(DIFFERENT.GT.TOL)THEN
         STOP TIMER
         GOTO 10
      END IF
```

```
      I=1
10    CONTINUE
      START TIMER
      ...
      CALL SOLVE()
      ...
      IF(DIFFERENT.GT.TOL)THEN
         STOP TIMER
         REDISTRIBUTE?
         GOTO 10
      END IF
```

```
      I=1
10    CONTINUE
      REDISTRIBUTE?
      START TIMER
      ...
      IF(...)THEN
         STOP TIMER
         GOTO 10
      ELSE
         GOTO 20
      END IF
      ...
20    CONTINUE
      ...
      IF(...)THEN
         STOP TIMER
         GOTO 10
      END IF
```

```
      I=1
10    CONTINUE
      START TIMER
      ...
      IF(...)THEN
         REDISTRIBUTE?
         STOP TIMER
         GOTO 10
      ELSE
         GOTO 20
      END IF
      ...
20    CONTINUE
      ...
      IF(...)THEN
         REDISTRIBUTE?
         STOP TIMER
         GOTO 10
      END IF
```

Figure 4.11: Placing the timers around the code containing the load imbalance, where REDISTRIBUTE? involves determining whether or not to redistribute the load + code to migrate the load.

In most instances, it makes no difference whether the load is balanced at the end of the current iteration or at the beginning of the next iteration, since the load is balanced before executing the next iteration based on the current level of load imbalance. The problem with balancing the load at the end of the loop is that this situation assumes that another iteration shall follow, and so the load can be redistributed unnecessarily on the last iteration. Whereas the problem with balancing the load at the beginning of the loop is that the load should not be

balanced on the first iteration, since the code within the loop has not yet been executed.

The decision to redistribute the load at the beginning of the loop was made due to several reasons. Firstly, the algorithm that determines when to redistribute the load will observe that there is no load imbalance on the first iteration, since the processor timings are the same, and in any case an IF statement could easily be used to guarantee that the load is not balanced here. Secondly, the load is balanced before executing the code for the current iteration, and so the load will never be redistributed unnecessarily in any kind of loop. Additionally, when balancing the load at the end of a DO Loop, it may be difficult to determine exactly which is the last iteration without explicitly specifying this value, as demonstrated in Figure 4.12, and so it may not be possible to stop the load from being balanced unnecessarily. With iterative loops the next iteration will always be performed after a possible redistribution. Finally, and more importantly, only one REDISTRIBUTE? section is needed if placed at the loop start, whereas several may be needed at the loop end (as illustrated in the last example shown in Figure 4.11), avoiding the need to insert the migration calls more than once.

```
DO I=1,NI
   IF(I.GT.1)THEN
      REDISTRIBUTE?
   END IF
   START TIMER
   ...
   CALL SOLVE()
   ...
   STOP TIMER
END DO
```

```
DO I=1,NI
   START TIMER
   ...
   CALL SOLVE()
   ...
   STOP TIMER
   IF(I.LT.NI)THEN
      REDISTRIBUTE?
   END IF
END DO
```

Figure 4.12: Example in which the load is not redistributed on the first or last iteration unnecessarily.

Better still, all of the inserted DLB code discussed in this Section can be placed together at the beginning of an iteration (Figure 4.13). The loop timers are stopped, the application may then be redistributed and then the loop timers are started again before performing the next iteration. Note that the condition that the load is not redistributed on the first iteration can be incorporated into the CAP_DLB_DECIDE utility along with the call to CAP_DLB_STOP_TIMER which stops timing the imbalanced loop.

```
DO I=1,NI
   REDISTRIBUTE?
   START TIMER
   ...
   CALL SOLVE()
   ...
END DO
```

```
      I=1
10    CONTINUE
      REDISTRIBUTE?
      START TIMER
      ...
      IF(...)THEN
         GOTO 10
      ELSE
         GOTO 20
      END IF
      ...
20    CONTINUE
      ...
      IF(...)THEN
         GOTO 10
      END IF
```

Figure 4.13: Redistribution only occurs at the beginning of an iteration.

4.5 Determine When To Redistribute

The user need not worry about when to redistribute the load, as this can be determined automatically within the CAP_DLB_DECIDE utility (Section 3.4 for more detail), which is placed at the start of the DLB Loop. The processor iteration times are obtained and a flag (CAP_DLB_PERFORM_REBAL) is set to True if it has been deemed necessary to redistribute the load before the next iteration.

4.6 Calculating The New Processor Partition Range Limits

The new distribution is only calculated if it has already been decided that redistributing the load may prove profitable. In order to obtain the new processor partition range limits for each processor a call to the utility CAP_DLB_START_REBAL is made, where the new limits are validated to determine whether the load needs to be migrated in a given dimension. CAP_DLB_MIGRATE_DIM is set to either True or False depending if the limits have changed in each dimension, indicating whether the load needs to be migrated in the given dimension.

4.7 Implementing The New Distribution

There are three distinct stages involved in the migration of data, where the first stage ensures processor ownership of data, the second stage updates the processor partition range limits, and the third stage involves updating the halo region. Ensuring that each processor owns a copy of the data defined by their new limits can be achieved using the migration utilities (Section 3.7), where a migration call needs to be constructed in each of the dimensions for every variable being migrated. The order in which the data is migrated is not significant, as it makes no difference whether the distribution is updated in the Left/Right direction followed by the Up/Down direction or vice versa. When manually implementing the DLB strategy, it has been decided that the data should be migrated in the order of partitioning (i.e. migrate the data in the first dimension, then second dimension, etc). Setting the order in which the data is migrated establishes a standard to follow, making the readability of the new DLB code easier for the user.

In order to fully implement the newly calculated distribution, each processor must own current and up-to-date values for all of the data that they may need to use, and not just the data in their defined workspace. Data in the halo region will also be needed, but this data has not been updated in the migration stage. Therefore the halo region must be updated on each processor if data within the halo region is to be used after redistribution. This must occur after the migration stage, and after having reassigned the processor partition range limits, so that the halo region is updated using the correct processor limits, and the correct data. Section 4.7.3 explains how to update the halo region in more detail.

4.7.1 Construct The Necessary Migration Calls

After calculating the new processor partition range limits the data needs to be migrated onto neighbouring processors using the migration calls described in Section 3.7 to update arrays, where data is migrated in each partitioned dimension separately. Partitioned data that is only used before redistribution (i.e. before the

DLB Loop) will not be affected by the change, but data used after redistribution will need to be migrated.

The first stage is to identify data that needs to be migrated, which can be done by looking for statements that occur after redistribution that involve partitioned and implicitly partitioned data (Section B.9.1.2), variables U, W, Y and Z in Figure 4.14. There is no easy way of identifying the partitioned data other than visually examining each statement. Nevertheless, it is possible to use the declarations (and Common Block statements) in the DLB Routine as a basis for identifying the partitioned data, since the partitioned data should be declared in the routine that uses it. The data declared in the DLB Routine will usually also contain the data that is used in called routines. These called routines will use the newly distributed data since it shall be executed after redistribution, meaning that most of the data in the called routines will already have been identified (reducing the effort needed). Arrays in SAVE statements may also need to be considered, where these arrays in called routines are not visible in the DLB Routine, making the manual implementation of the DLB strategy very difficult.

The migration calls will be grouped separately according to the Migration Dimension (see Figure 4.15), but this process is further complicated since arrays may be 'partitioned' in none, some or all dimensions. It is easy to fail to identify data that needs to be migrated, where this will become obvious during execution, in which case the user will have to examine the code again more closely.

S1	SUBROUTINE DLBROUTINE(X,Z,…)	*X is not used after redistribution, unlike Z at S14*
S2	DECLARATIONS	*Declarations should contain U, V, W, X, Y, and Z*
S3	COMMON BLOCKS	*Common block may contain these variables*
	…	
S4	U( )=	*U is used after redistribution at S11*
S5	V( )=	*V is not used after redistribution*
S6	W( )=	*W is used after redistribution at S13*
S7	Y( )=	*Y is used after redistribution at S14*
	…	
S8	…=V( )+Y( )+X( )	
	…	*Usage before redistribution ↑*
S9	DO DLBLOOP	
S10	*REDISTRIBUTION*	
	…	*Usage after redistribution ↓*
S11	…=**U**( )	
	…	
S12	END DO	
	…	
S13	…=**W**( )	
	…	
S14	CALL SUB1(**Y**,**Z**,…)	
	…	
S15	END	

Figure 4.14: Partitioned data that is used after redistribution will need to be migrated.

In order to use the migrated data inside this routine this data must be declared in the DLB Routine if it has not already been declared. This may involve introducing a COMMON block to replace SAVE statements in called routines. Once this has been done, the low declared limit in the declaration statement can be used, along with the lower processor partition range limit, to construct the starting address of the migrated data. The lower processor partition range limit is used in all Non-Staggered Dimensions apart from the Migration Dimension, for which a new starting address will be offset from the low declared limit. All other indices, including the staggered index, will use the low declared limit, an example of which can be seen in Section 3.7.1.

The second stage is to determine what type of call is going to be required to migrate this data variable (Section 3.7). There are two types of migration calls, one that communicates data to an immediate neighbour (CAP_MIGRATE), and one that migrates data to several neighbours using several internal

communications (CAP_DLB_MIGRATE). If the data is not also partitioned in the Staggered Dimension, then CAP_MIGRATE can be used, as these communications are not affected by the staggered limits. When migrating in a Non-Staggered Dimension in which the data has also been partitioned in the Staggered Dimension then CAP_DLB_MIGRATE is needed. In terms of construction, there are only minor differences between these two types of calls, as CAP_MIGRATE is to some extent a subset of CAP_DLB_MIGRATE, which has two additional parameters relating to the Staggered Dimension included in the parameter list. A check to see if the data is partitioned in the Staggered Dimension therefore needs to be made if migrating in a Non-Staggered Dimension.

real
U(2:7, 3:12, 4:6, 5:19, 6:9, 7:16, 8:15)
2nd 3rd 1st

Index	L	Stride	Length
1	2	1	6
2 (2nd)	**3**	**6**	10 (H2-L2+1)
3	4	60	3
4 (3rd=SD)	**5**	**180**	15 (H3-L3+1)
5	6	2700	4
6 (1st)	**7**	**10800**	10 (H1-L1+1)
7	8	108000	8

Where L1=CAP1_LOW H1=CAP1_HIGH
and L2=CAP2_LOW H2=CAP2_HIGH

integer
W(2:7, 3:12, 4:6, 6:9, 7:16, 8:15)
2nd 1st

Index	L	Stride	Length
1	2	1	6
2 (2nd)	**3**	**6**	10 (H2-L2+1)
3	4	60	3
4	6	180	4
5 (1st)	**7**	**720**	10 (H1-L1+1)
6	8	7200	8

Index 3 and 4 can be paired

```
IF(MIG_DIM(1))THEN
   CALL CAP_DLB_MIGRATE(U(2,L2,4,5,6,7,8),7,10800,5,180,1,6,
                                    6,H2-L2+1,60,3,2700,4,108000,8,1,1,2,1)
   CALL CAP_MIGRATE(W(2,L2,4,6,7,8),7,720,1,6,6,H2-L2+1,60,12,
                                  7200,8,1,1,1,1,1,1)
   ...
END IF
IF(MIG_DIM(2))THEN
   CALL CAP_DLB_MIGRATE(U(2,3,4,5,6,L1,8),3,6,5,180,1,6,60,3,2700,4,
                                    10800,H1-L1+1,108000,8,1,1,2,2)
   CALL CAP_MIGRATE(W(2,3,4,6,L1,8),3,6,1,6,60,12,720,H1-L1+1,
                                  7200,8,1,1,1,1,1,2)
   ...
END IF
IF(MIG_DIM(3))THEN
   CALL CAP_MIGRATE(U(2,L2,4,5,6,L1,8),5,180,1,6,6,H2-L2+1,60,3,
                                  2700,4,10800,H1-L1+1,108000,8,2,3)
   ...
END IF
```

Figure 4.15: Construction of migration calls using information relating to the migrated data.

In order to construct the migration call the data type and the processor axis (Migration Dimension number) are needed, along with the start index (low declared limit) and stride for the Migration Dimension and the Staggered Dimension (if using CAP_DLB_MIGRATE). The data type is set according to the declaration type using the CAPTools standard (Table A.1). The Migration Dimension is simply the number representing the pass in which the data is being migrated, where 1 is used when migrating in the first partitioned dimension, 2 is used when migrating in the second partitioned dimension, etc. The stride (S_i) and number of strides (NS_i) for each remaining index or group of contiguous indices are also needed to construct this migration call. More data can be buffered within the migration call if indices are grouped together into continuous sections of data, where 1 is used if there are remaining indices than there are parameters (as there are 6 S/NS pairs inside a call). The stride of the first paired-index is the stride for the first index in that group, and the number of strides is the length of the continuous data (the product of the dimensions) of the indices in this group. If the data is partitioned (in a Non-Staggered Dimension other than the Migration Dimension) then the stride is the partition stride, and the number of strides is the length of the partition (CAP_HIGH-CAP_LOW+1).

4.7.2 Reassign The Limits

Up until now the new processor partition range limits have only been calculated and used within the new DLB code, but have never actually been set up for use in the parallel code. The new limits therefore need to be updated for use within the code using the CAP_DLB_REASSIGNLOWHIGH utility (Section 3.9), as demonstrated in Figure 4.16. The new limits are used in the migration of data in subsequent dimensions to ensure they move the correct amount of data (Section 2.6). In addition, the DLB strategy cannot be implemented properly without updating the values of the processor partition range limits (where the old limits are currently being used).

The internal processor partition range limits also need to be updated for use inside the DLB utilities, which is achieved by calling the

CAP_DLB_NEW2OLD_LIMITS utility (Section 3.9) after the load has been migrated in all dimensions. These internal limits need to be updated before any other DLB utilities are called, such as the DLB communications used to update the halo region in the next Section.

```
IF(MIG_DIM(1))THEN
   all of the migration calls used to migrate in the 1st partitioned dimension
   CALL CAP_DLB_REASSIGNLOWHIGH(CAP1_LOW,CAP1_HIGH,1)
END IF
IF(MIG_DIM(2))THEN
   all of the migration calls used to migrate in the 2nd partitioned dimension
   CALL CAP_DLB_REASSIGNLOWHIGH(CAP2_LOW,CAP2_HIGH,2)
END IF
IF(MIG_DIM(3))THEN
   all of the migration calls used to migrate in the 3rd partitioned dimension
   CALL CAP_DLB_REASSIGNLOWHIGH(CAP3_LOW,CAP3_HIGH,3)
END IF
IF(MIG_DIM(1) .OR. MIG_DIM(2) .OR. MIG_DIM(3))THEN
   CALL CAP_DLB_NEW2OLD_LIMITS
END IF
```

Figure 4.16: The processor partition range limits of a particular dimension are updated using CAP_DLB_REASSIGNLOWHIGH after migrating the load in that dimension, after which CAP_DLB_NEW2OLD_LIMITS is used to update the internal processor partition range limits used in the DLB utilities.

4.7.3 Update The Halo Region After Redistribution

The migration calls mentioned in the previous Section are used to ensure that each processor owns the data defined by its new processor partition range limits. These calls only guarantee the use of current data values within the processor partition range limits and not within the halo region, which is also required. Figure 4.17 illustrates the use of halo data after redistribution, indicating the need to update data in the halo region as well as within the processor partition range limits, as illustrated graphically in Figure 4.18. If the data in the halo region were not updated then this would lead to an incorrect solution to the problem, since old (or uninitialised) values would be used. This suggests that a communication call is needed to update the halo region, using the new processor partition range limits in every partitioned dimension.

```
DO J=...
   DO I=...
      U(I,J)=...
   END DO
END DO
```

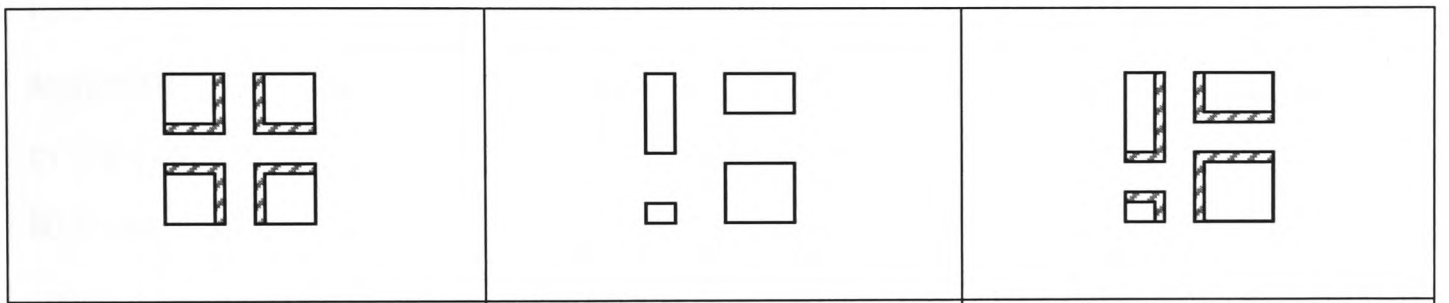
Communication to update the halo region for U

```
DO N=1,NSTEP
   REDISTRIBUTE?
   START TIMER
   ..........
   ...=U(I,J)+U(I+1,J+1)+...
   ..........
END DO
DO J=...
   DO I=...
      ...=U(I,J)+U(I-1,J-1)+...
   END DO
END DO
```

Usage's of communicated data using original distribution

Halo unknown – not updated by migration calls

Figure 4.17: Code extract showing usage of halo data after redistribution.



Current distribution in which the halo region has been updated for use.	New distribution after migration, where the halo region is unknown.	Workload after updating the halo region.

Figure 4.18: Illustration showing the need to update the halo region after data migration.

There are several possible solutions to updating the halo region with the newly distributed data. For example, the first is to create new overlap communications from scratch using any existing dependency information. The second solution is to incorporate the halo region into the migration call itself, where the width of the halo region is passed in. Finally, existing halo communications that occur before redistribution could be duplicated.

In terms of manually implementing this DLB Staggered Limit Strategy, the easiest option to implement from those given above would be solution (3), in which existing communications are simply duplicated after updating all of the processor partition range limits. The second solution means ensuring that the correct halo width is incorporated into each migration call, which needs to be carefully calculated i.e. the user needs to look for the largest halo width (e.g. U(I+4)) that shall be used after redistribution, some of which may be subsets of

others. Additionally, the halo region may not be used by all of the migrated data, and it is unnecessary to extend the width of the communicated message in the migration call. One problem in using this approach is that data may be assigned in the halo region, complicating this task further since more effort is required to identify the halo width to be incorporated. By duplicating existing communications (solution 3), the user does not need to worry about having to generate the correct code, since much testing has already been done pre-DLB, incorporating the correct overlap width into the call, which means that the user simply has to ensure that the correct communication is duplicated.

There is no need to examine every statement of the application code, as only those halo communications that occur before redistribution are important. Those halo communications that occur after redistribution (within or below the DLB Loop) will use the current data values that have recently been migrated, suggesting these communications can be ignored. In Figure 4.19 for example, all of the halo communications between the DLB Loop Head and the end of the DLB Routine will use the newly migrated data, and the same applies to those halo communications that are executed after the call to Sub_DLB in the Main program. All of the statements that are executed before the DLB Loop need to be examined, where every statement in a called routine will also need to be examined. In Figure 4.19 for example, every statement between the start of the DLB Routine and the DLB Loop need to be examined along with every statement in Sub_2. Additionally, every statement between the routine start and the call to the DLB Routine need to be examined for calling routines (callers of the DLB Routine). For example, every statement between the start of the Main program and the call to Sub_DLB need to be examined, along with every statement in Sub_1. Figure 4.20 illustrates how to identify those communications that need to be duplicated.

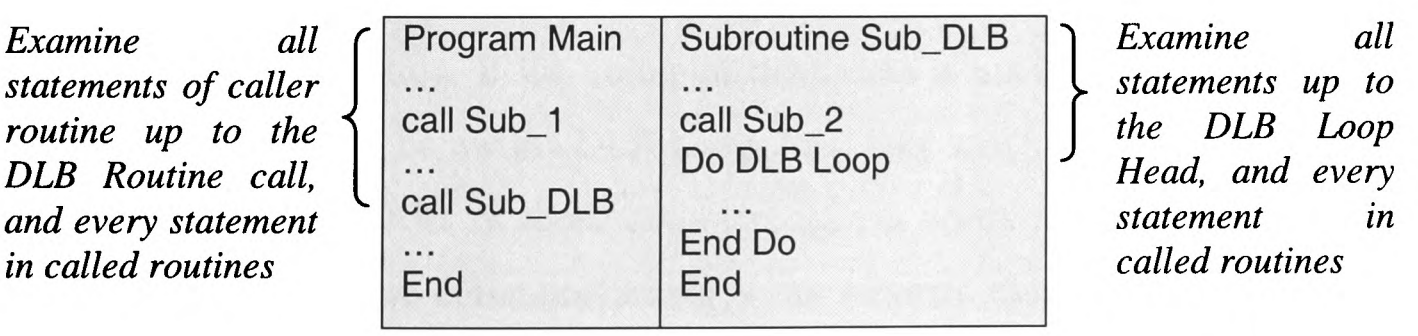


Figure 4.19: Statements executed before redistribution need to be examined for halo communications that may be duplicated.

S1	X( )=	
S2	**Comm X( ) halo**	do not duplicate because data not used after redistribution
S3	=X( )	
S4	Y( )=	
S5	**Comm Y( ) halo**	duplicate because data used in S17 after redistribution
S6	=Y( )	
S7	Z( )=	
S8	**Comm Z( ) halo**	duplicate because data used in S23 after redistribution
S9	V( )=	
S10	**Comm V( ) halo**	duplicate because data used in S18 and S24 after redistribution
S11	P( )=	
S12	T( )=	
S13	DO DLB Loop	
S14	*REDISTRIBUTE?*	
S15	**Comm P( ) halo**	do not duplicate because halo is updated with new values
S16	**Comm T( ) halo**	do not duplicate because halo is updated with new values
S17	=Y( )	
S18	=V( )	
S19	P( )=P( )+…	
S20	=T( )	
S21	END DO	
S22	=T( )	
S23	=Z( )	
S24	=V( )	
S25	W( )=	
S26	**Comm W( ) halo**	do not duplicate because halo is updated with new values
S27	=W( )	

Figure 4.20: Illustration showing how to identify communications that need to be duplicated. Communications occurring after redistribution do not need to be duplicated, as these communications use the newly updated data distribution.

Therefore, any halo communications that occur before redistribution including those in the routines that call the DLB Routine, will need to be considered for duplication. If the communicated data is used after redistribution, which means within or below the DLB Loop, then they will have to be duplicated, where the execution order of these communications needs to be retained. If the execution order of these communications is not retained then the communicated data may be incorrect, as a communication may be updating an extended halo region with a value that is presumed as having been updated in a previous

dimension (Figure A.10). For example, the Left/Right communications are always executed before any Up/Down communications where the Up/Down communications can include the Left/Right halo region, and so this order of execution needs to be retained otherwise some out-of-date values may be communicated (Figure 4.21).

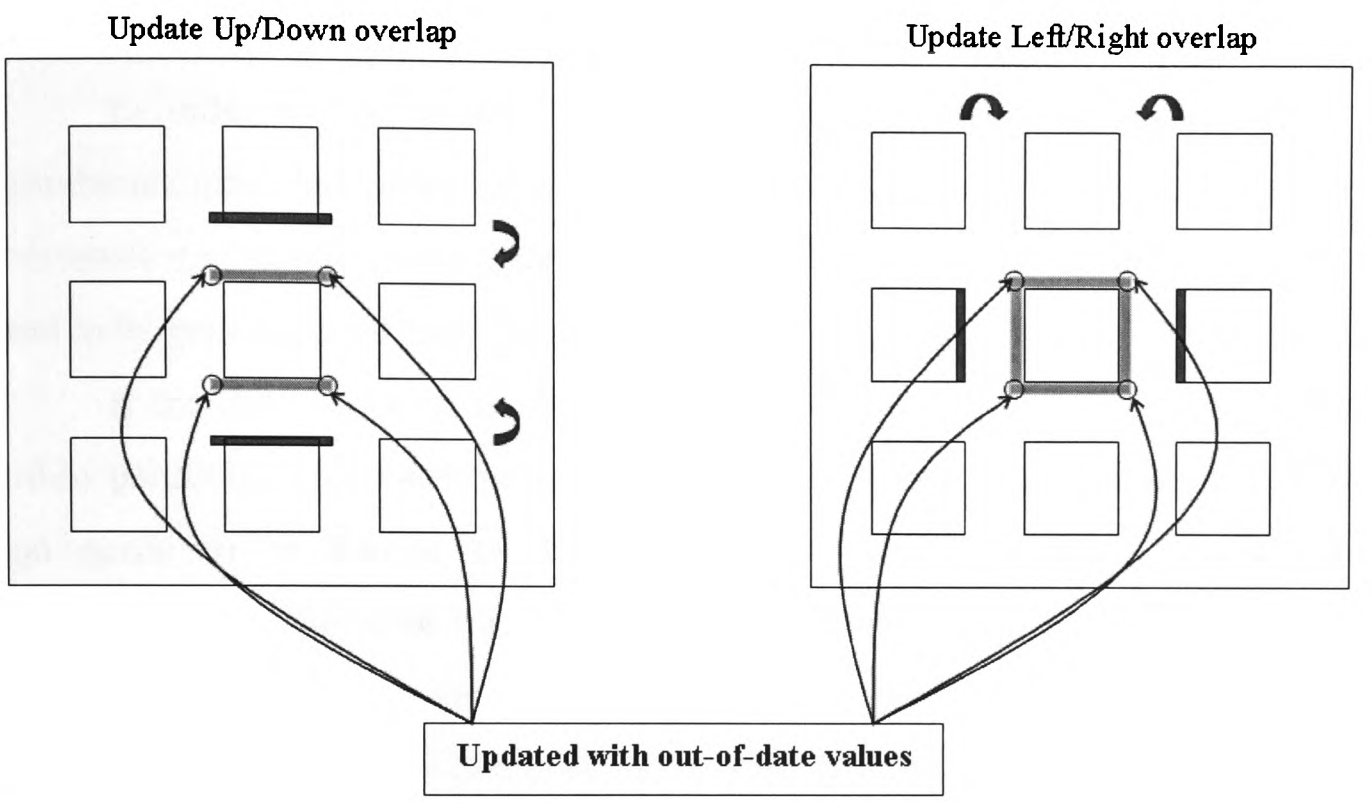


Figure 4.21: Result when communications are duplicated with no regard to their order of execution. When duplicates of Up/Down communications are placed before Left/Right communications then out-of-date values are used.

The halo region of unpartitioned arrays will also need to be updated using this approach, since communications of these arrays will be duplicated along with those of any partitioned arrays. A problem arises when the halo region is assigned on a processor and not communicated, as there will be no communication to duplicate, as demonstrated in Figure 4.22. In this situation the user must look for any assignments involving data in the halo region. Nothing needs to be done if the assigned halo region is not used after redistribution, but if this halo region is used after redistribution then a new communication must be constructed for this data, where the halo data is communicated with a neighbouring processor.

In Figure 4.22, both KPLUS and KMINUS are partitioned in the second pass, and are initialised once at the beginning of the code, and so there are no halo communications. After redistribution the values of KPLUS and KMINUS still need to be known.

```
DO K=MAX(1,CAP2_LOW-2),MIN(KMAX,CAP2_HIGH+1),1
   KPLUS(K)=K+1
   IF( (K.LE.CAP2_HIGH .AND. K.GE.CAP2_LOW) .OR.
       ((K-1).LE.CAP2_HIGH .AND. (K-1).GE.CAP2_LOW) .OR.
       ((K+1).LE.CAP2_HIGH .AND. (K+1).GE.CAP2_LOW) )THEN
      KMINUS(K)=K-1
   END IF
END DO
```

Figure 4.22: Example from ARC2D in which there is no halo communication to duplicate, since the halo region is initially assigned on each processor.

In order to update the halo region the user must search through all communications that occur before redistribution, duplicating any in which the communicated data is used after redistribution. Similarly, new communications need to be generated for assigned halo regions that are used after redistribution.

If the data is not an array but a scalar, whose value is assigned and used within particular processor partition range limits (Figure 4.23), then this scalar also needs to be owned by the new owner of the processor limits after redistribution. In this case the execution control mask of the assigned scalar can be used in constructing a 'special' migration call, in which the current owner only passes this data to the new owner of the specified limits. The new owner of row 8 needs to know the value of P which was assigned on the old owner of row 8.

```
IF (8.LE.CAP2_HIGH .AND. 8.GE.CAP2_LOW)
   P=50
END IF
...
Redistribution
IF (8.LE.CAP2_HIGH .AND. 8.GE.CAP2_LOW)
   ...=P
END IF
```

Figure 4.23: Example illustrating the need to migrate a scalar variable that is assigned and used between given processor partition range limits.

4.7.3.1 Identifying Potential Communications To Duplicate

The user must search through the relevant code for any communications involving data in the halo region, examining those communications that may potentially be duplicated. If the communication is a Broadcast or a Commutative then this

implies that every processor will have a copy of the communicated data, in which case there is no need to duplicate such a communication. The user must be aware that if the halo communication is contained within a DO Loop, or IF, construct along with other executable code then this particular halo communication no longer needs to be duplicated. The reasoning behind this is that if, for example, a halo communication is positioned at the top of a loop containing other executable statements then this implies that the communication has been placed there simply for use by the other statements within the loop due to the CAPTools migration algorithm (Section B.9.1.4). As mentioned in Section A.3.3, if the communication was simply buffered within a DO Loop then the communication would be the only executable statement within it (including any masks), in which case the surrounding DO Loop construct must also be duplicated. Similarly, when a communication is contained within an IF construct then if it is the only executable statement then the communication and surrounding construct(s) will have to be duplicated, otherwise it need not be considered for duplication. Again, if there are any other executable statements in the IF construct then this means that the halo communication will be used by these statements, otherwise the communication would have been migrated higher up in the code. Note that if the DO or IF construct contains another DO or IF construct then the same rule applies, where the communication should only be duplicated if there are no other executable statements within the construct. Similarly, communications contained within an IF ELSE construct can be ignored because these will not be halo communications, but they will be communications specific to the executable code that follows within the construct.

In order to use the duplicated communication (and any surrounding construct) inside the DLB Routine, then all variables needed for the communication statement must be declared in the DLB Routine if it has not already been declared.

4.8 Example DLB Code

The following example, shown in Figure 4.24, is used to demonstrate how to implement the DLB Staggered Limit Strategy within a parallel Jacobi code, where the DLB code is highlighted. In this example there is only one routine, and so all of the DLB code is contained within this routine. The arrays were first partitioned in the second index, and then the first, where the communications in the Up/Down direction are in the Staggered Dimension, meaning that only communications in the Left/Right direction may need to be changed into DLB communications. The DLB timer code surrounds the iteration loop, where migration calls are generated in each partitioned dimension for the two arrays (T and TNEW). After migrating the data in every partitioned dimension, all halo communications that occur above redistribution, whose data is used after redistribution, are duplicated after updating the processor partition range limits. These duplicated communications can be for any partitioned dimension and not just for the Non-Staggered Dimensions. Note that additional code involving the arrays V and X has been added to demonstrate overlap updating.

```
      REAL T(0:1001,1000),TNEW(1000,1000),V(1000,1000),X(1000,1000)
S1    DECLARATIONS
S2    INITIALISATIONS
      C Store the potential neighbours of each processor
S3    CALL CAP_DLB_SETALLNEIGHBOURS
      C Read in N – the square grid size of this 2D Jacobi code
      C Set up processor partition range limits - store processor limits for each potential
        neighbour
S4    CALL CAP_SETUPDPART(1,N,CAP2_LOW,CAP2_HIGH,2)
S5    CALL CAP_DLB_SETUPLIMITS(CAP2_LOW,CAP2_HIGH,2)
S6    CALL CAP_SETUPDPART(1,N,CAP1_LOW,CAP1_HIGH,1)
S7    CALL CAP_DLB_SETUPLIMITS(CAP1_LOW,CAP1_HIGH,1)
S8    CALL CAP_DLB_EXCHANGE(X(CAP2_LOW,CAP1_LOW-1),X(CAP2_LOW,
                            CAP1_HIGH),CAP2_HIGH-CAP2_LOW+1,
                            CAP2_LOW,1,CAP2_LOW,CAP2_HIGH,
                             2,CAP_LEFT)
      NITER=0
      C Stop timing the imbalanced loop iteration
      C Decide whether dynamic load balancing is profitable – based on timings obtained
S9    40 CALL CAP_DLB_DECIDE(CAP_DLB_WALLTIME,CAP_DLB_COMMTIME,
                             CAP_DLB_COMPTIME,CAP_DLB_MAXTIME,
                             CAP_DLB_PERFORM_REBAL,CAP_DLB_ITER,
                             CAP_DLB_REBAL_ITER,CAP_DLB_REBALTIME)
S10   IF (CAP_DLB_PERFORM_REBAL) THEN
      C   DLB is profitable
      C   Find new processor partition range limits – justify load migration
```

```
S11       CALL CAP_DLB_START_REBAL(CAP_DLB_REBALTIME,
                                   CAP_DLB_COMPTIME,
                                   CAP_DLB_MAXTIME,
                                   CAP_DLB_PREV_REBALTIME,
                                   CAP_DLB_MIGRATE_DIM)
S12       IF (CAP_DLB_MIGRATE_DIM(1)) THEN
        C   Load migration in the first partitioned dimension(Non-Staggered) is justified –
            enough load to migrate
S13         CALL CAP_DLB_MIGRATE(T(0,1),1,1002,0,1,1,1,1,1,1,1,1,1,1,1,1,1,2,1)
S14         CALL CAP_DLB_MIGRATE(TNEW(1,1),1,1000,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                                 2,1)
        C   Reassign processor limits
S15         CALL CAP_DLB_REASSIGNLOWHIGH(CAP1_LOW,CAP1_HIGH,1)
          END IF
S16       IF (CAP_DLB_MIGRATE_DIM(2)) THEN
        C   Load migration in the second partitioned dimension (Staggered) is justified –
            enough load to migrate
S17         CALL CAP_MIGRATE(T(0,CAP1_LOW),0,1,1002,CAP1_HIGH-CAP1_LOW
                             +1,1,1,1,1,1,1,1,1,1,1,2,2)
S18         CALL CAP_MIGRATE(TNEW(1,CAP1_LOW),1,1,1000,CAP1_HIGH-
                             CAP1_LOW+1,1,1,1,1,1,1,1,1,1,1,2,2)
        C   Reassign processor limits
S19         CALL CAP_DLB_REASSIGNLOWHIGH(CAP2_LOW,CAP2_HIGH,2)
          END IF
S20       IF (CAP_DLB_MIGRATE_DIM(1) .OR. CAP_DLB_MIGRATE_DIM(2)) THEN
        C   Some cells have been migrated - update stored processor partition range
            limits
S21         CALL CAP_DLB_NEW2OLD_LIMITS
        C   Duplicate overlap communications (if any)
S22         CALL CAP_DLB_EXCHANGE(X(CAP2_LOW,CAP1_LOW-1),
                                  X(CAP2_LOW,CAP1_HIGH),
                                  CAP2_HIGH-CAP2_LOW+1,CAP2_LOW,1,
                                  CAP2_LOW,CAP2_HIGH,2,CAP_LEFT)
S23         CALL CAP_DLB_STOP_REBAL(CAP_DLB_REBALTIME,CAP_DLB_ITER,
                                    CAP_DLB_REBAL_ITER)
          ELSE
S24         CAP_DLB_REBALTIME=CAP_DLB_PREV_REBALTIME
          END IF
        END IF
        C   Start timing the imbalanced loop iteration
S25     CALL CAP_DLB_START_TIMER(CAP_DLB_WALLTIME,
                                 CAP_DLB_COMMTIME,
                                 CAP_DLB_COMPTIME)
        NITER=NITER+1
        DIFFMAX=0.0
        CALL CAP_DLB_EXCHANGE(T(CAP2_LOW,CAP1_LOW-1),T(CAP2_LOW,
                              CAP1_HIGH),CAP2_HIGH-CAP2_LOW+1,
                              CAP2_LOW,1,CAP2_LOW,CAP2_HIGH,
                              2,CAP_LEFT)
        CALL CAP_DLB_EXCHANGE(T(CAP2_LOW,CAP1_HIGH+1),T(CAP2_LOW,
                              CAP1_LOW),CAP2_HIGH-CAP2_LOW+1,
                              CAP2_LOW,1,CAP2_LOW,CAP2_HIGH,
                              2,CAP_RIGHT)
        CALL CAP_BEXCHANGE(T(CAP2_LOW-1,CAP1_LOW),T(CAP2_HIGH,
                           CAP1_LOW),1,1001,CAP1_HIGH-CAP1_LOW+1,
                           2,CAP_UP)
        CALL CAP_BEXCHANGE(T(CAP2_HIGH+1,CAP1_LOW),T(CAP2_LOW,
                           CAP1_LOW),1,1001,CAP1_HIGH-CAP1_LOW+1,
```

```
                                    2,CAP_DOWN)
      DO 60 J=MAX(2,CAP2_LOW),MIN(N-1,CAP2_HIGH),1
        DO 20 I=MAX(2,CAP1_LOW),MIN(N-1,CAP1_HIGH),1
          TNEW(J,I)=(T(J,I-1)+T(J,I+1)+T(J-1,I)+T(J+1,I))/4.0
          V(J,I)=V(J,I)+X(J,I)+X(J,I-1)
20      CONTINUE
60    CONTINUE
      DO J=MAX(2,CAP2_LOW),MIN(N-1,CAP2_HIGH),1
        DO I=MAX(2,CAP1_LOW),MIN(N-1,CAP1_HIGH),1
          T (J,I)=TNEW(J,I)
        END DO
      END DO
      DIFFMAX=
      IF (DIFFMAX.GT.TOL) THEN
        GOTO 40
      END IF
```

S1	*Declare inserted variables (e.g. CAP_DLB_WALLTIME).*
S2	*Initialise inserted variables (e.g. CAP_DLB_WALLTIME=0.0).*
S3	*Set up the processor topology array, containing the neighbouring processors of each processor in every partitioned dimension.*
S5, S7	*Set up the array containing the processor partition range limits in every partitioned dimension.*
S9	*Stop timing the iteration loop that contains the load imbalance and determine when the next redistribution will occur.*
S10	*Balance the load if the next redistribution is to occur on the current iteration.*
S11	*Start timing the load balancing process. Determine the new load distribution and decide if it is worth implementing..*
S12	*Decide to use the new distribution in this Non-Staggered Dimension.*
S13, S14	*Migrate the workload of the variable in this Non-Staggered Dimension.*
S15	*Reassign the new processor partition range limits in this Non-Staggered Dimension.*
S16	*Decide to use the new distribution in the Staggered Dimension.*
S17, S18	*Migrate the workload of the variable in the Staggered Dimension.*
S19	*Reassign the new processor partition range limits in the Staggered Dimension.*
S20	*Determine whether the stored processor partition range limits need to be updated.*
S21	*Update the processor partition range limits for use in DLB utilities.*
S22	*Duplicates of any halo communications in which the halo data is used after redistribution (e.g.: S8).*
S23	*Stop timing the redistribution.*
S24	*Reset the redistribution time to that of the previous redistribution, as no redistribution occurred (no load migration was necessary).*
S25	*Start timing the iteration loop that contains the load imbalance.*

Figure 4.24: Shows an extract of sample code in which the highlighted code represents the DLB code that has been inserted into it, and a brief explanation of the inserted statements.

In most large application codes, the number of DLB statements compared to the original parallel statements would be relatively small. Although the example given above is simple it should be noted that most of the changes to the user's code (calls other than DLB communications) are grouped together inside the DLB Loop in the actual load balancing section, and that most of these inserted statements run onto several lines.

4.9 Results And Observations

Due to time constraints, the manual implementation of the DLB Staggered Limit Strategy was only undertaken in a few codes, as the purpose of this research was to develop a DLB strategy that could be automated within CAPTools, and not simply to implement a strategy within as many codes as possible. The applications discussed here were used in a trial and error process to test that the DLB Staggered Limit Strategy and its utilities operated as expected.

4.9.1 The JACOBI Code

The JACOBI code is a very basic structured mesh application code. Using an explicit Jacobi solver with a 5-point-stencil, the 2D serial application consists of 37 lines of code. The CAPTools generated version of this code, in which index 2 and then index 1 have both been partitioned, consists of 98 lines of code, where the computational load is the same on every processor (i.e. it is physically balanced).

This simple code is ideal for testing the functionality of the DLB communications and so initial testing was done to ensure that the underlying operations of these newly devised communications were correct. Initial testing therefore involved converting some of the communications into DLB communications and then executing the code using non-coincidental limits in one of the partitioned dimensions. The functionality of the DLB communications were

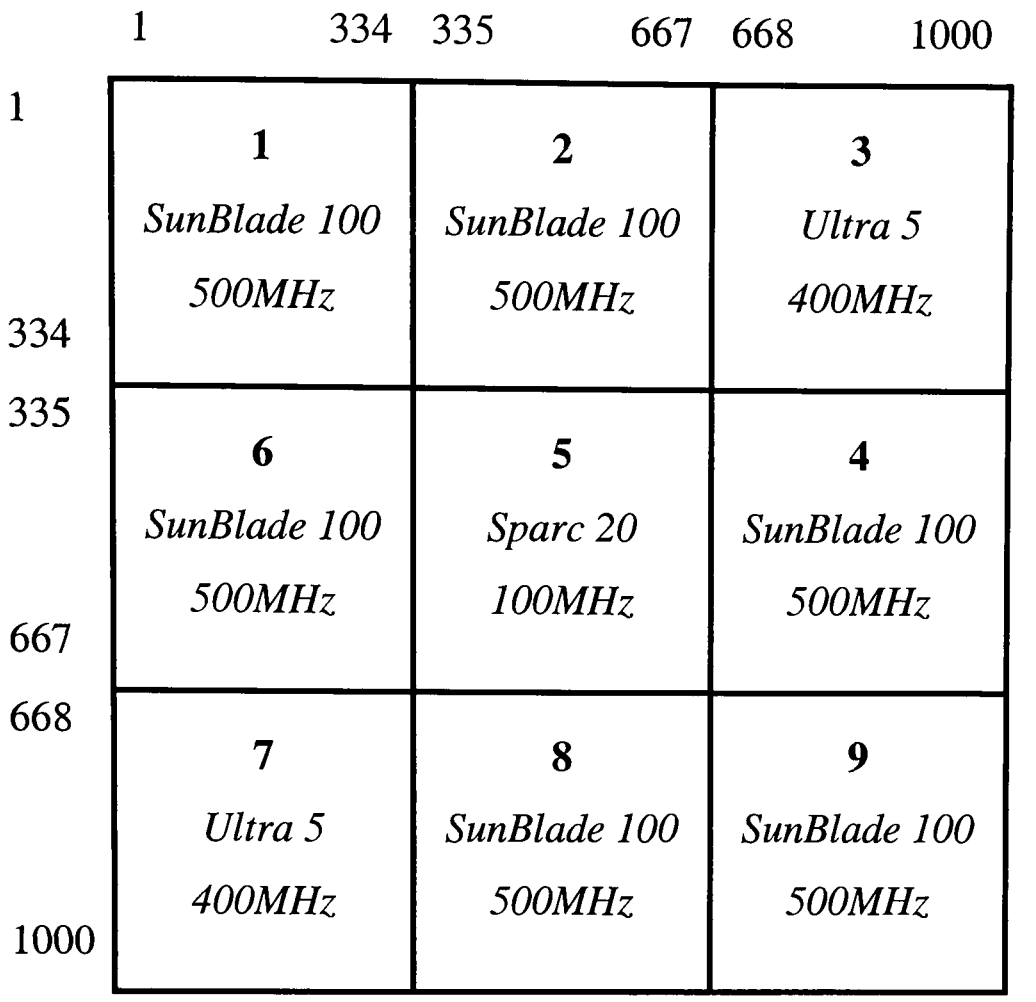
tested on numerous staggered partitions by changing the processor partition range limits of each processor manually at the start of execution (without the need for any load migration).

This application code was also used to test the algorithm that calculated the new workload distribution, enabling the investigation of the effects of dynamic load balancing assuming processor imbalance. Note that the code was slightly modified to highlight the differences between processors (the main loop in the code was repeated 150 times), as the timing of a single iteration did not show any significant difference worth reporting.

Figure 4.25 shows the initial distribution along with the computation times (the difference between the wallclock time and communication time) for the first 150 iterations of a 1000x1000 JACOBI mesh code mapped onto a 3x3 heterogeneous processor topology. Due to availability, the system consists of six 500MHz SunBlade 100 processors, two 400MHz Ultra 5 processors and one 100MHz Sparc 20 processor in the middle. In this problem the load imbalance is due to processor imbalance, as each processor has the same computational workload but only differs in speed and the number of users. Therefore the algorithm used to determine the new distribution assumes that gained cells will be processed at the weight of the gaining processor.

The middle processor is the slowest and so it is clear that the load on this processor needs to be reduced quickly, as a large amount of idle time accumulates on the surrounding processors. Continuing with the initial distribution would be detrimental because the computation time of the slowest processor (58.32 seconds) would dominate the iteration time, and so the other processors would continue to be idle for up to approximately 47 seconds (the difference between the fastest and slowest processor). Note that Processors 3 and 4 are slightly slower than the SunBlade 100's, and so ideally these processors should end up with slightly less work than the SunBlades after redistribution, although they should have more work than the Sparc 20.

Iteration 1 (initial distribution):



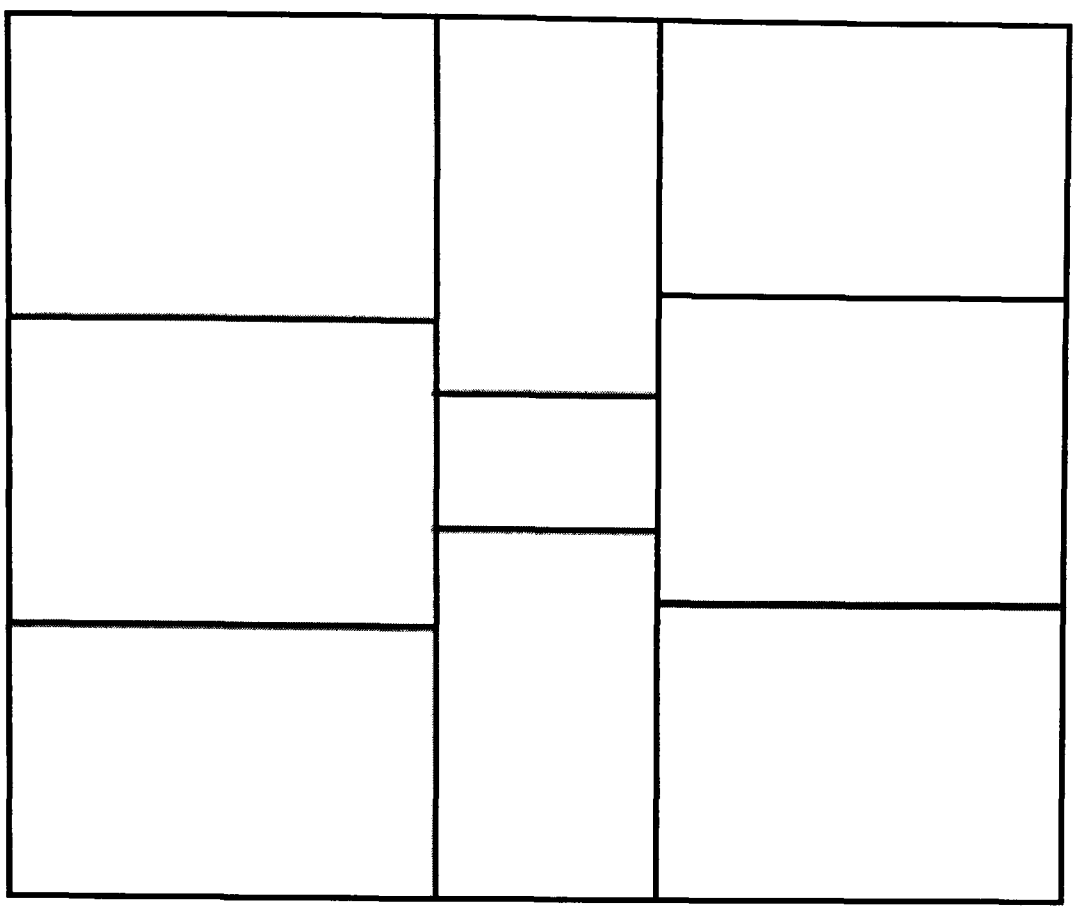
Processor	Timing	Left	Right	Up	Down	Workload
1	11.53	1	334	1	334	111556
2	11.66	335	667	1	334	111222
3	13.93	668	1000	1	334	111222
4	11.57	668	1000	335	667	110889
5	**58.32**	**335**	**667**	**335**	**667**	**110889**
6	11.54	1	334	335	667	111222
7	13.42	1	334	668	1000	111222
8	11.44	335	667	668	1000	110889
9	11.74	668	1000	668	1000	110889

Figure 4.25: The processor timings and processor partition range limits of the first iteration for a heterogeneous 3x3 processor topology (based on a cluster of workstations) that has been mapped evenly onto a 1000x1000 JACOBI mesh code application.

With a ratio of 3.38 (the maximum processor time divided by the average time), the load is first redistributed at the beginning of iteration 2 with a time of 2.88 seconds, where the new distribution, processor timings, partition range limits and workloads using this partition are shown in Figure 4.26. After a single redistribution the system is already 3 times faster with a maximum processor

timing of 17.38 seconds, a reduction of 40.94 seconds. Although Processor 5 now has a workload of 32494 cells, Processors 2 and 8 do not have enough work to process. Therefore, with a ratio of 1.27 the second redistribution occurs at the beginning of iteration 3 with a time of 0.59 seconds with Processors 2 and 8 gaining cells whilst Processor 5 loses some more of its workload (Figure 4.30). Having a maximum processor timing of 16.53 seconds and a ratio of 1.22, the third redistribution occurs at the beginning of the fourth iteration with a time of 0.36 seconds (Figure 4.31). The processor workloads for the fourth iteration appear to reflect the differences between the processors used in the system, with the SunBlade 100's typically having more work than the Ultra 5's who have more work than the Sparc 20. With a maximum processor timing of 13.86 seconds the system seems to be sufficiently well balanced after three redistributions (approximately 4 times faster than when using the initial distribution). Continuing with this distribution, the ratio of the processor timings for iteration 16 (Figure 4.29) is still 1.04 (the same as for iteration 4) with a maximum processor timing of 13.93 seconds, indicating that the system is stabilising somewhat.

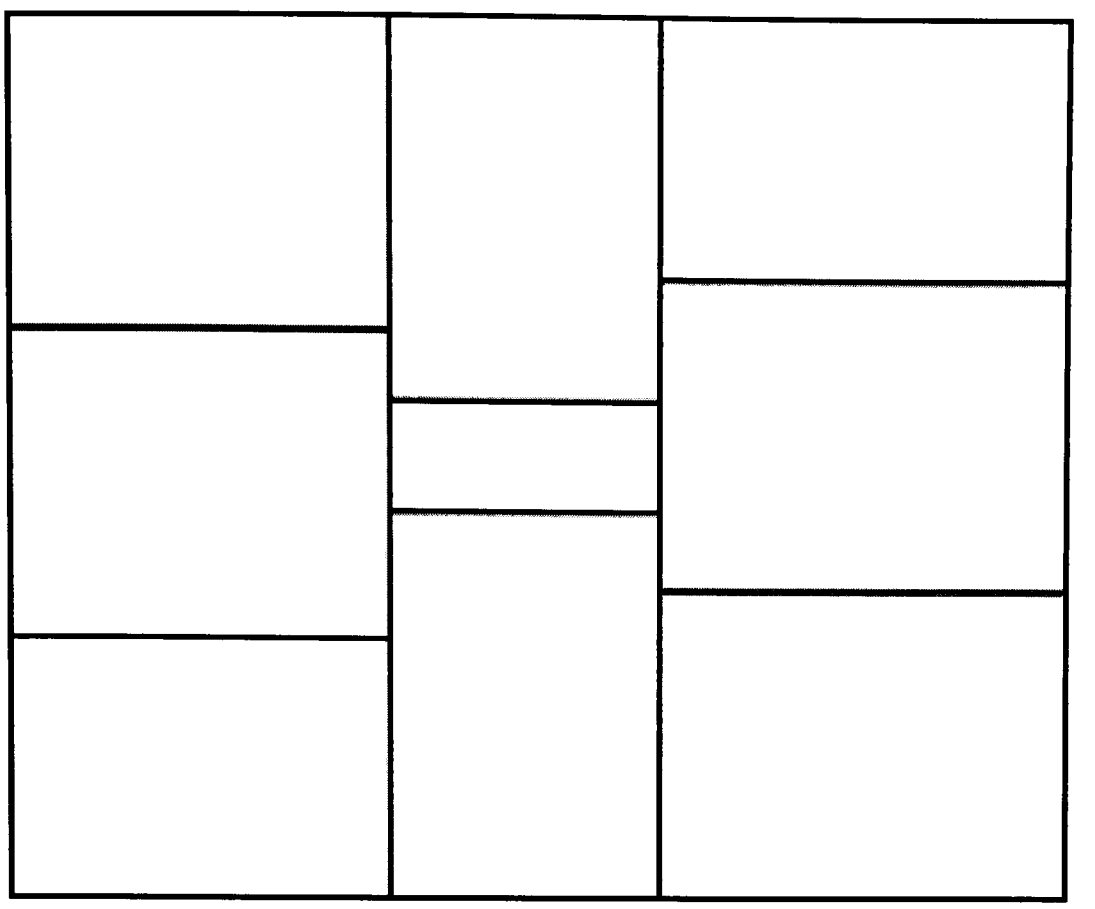
Iteration 2:



Processor	Timing	Left	Right	Up	Down	Workload
1	14.56	1	399	1	345	137655
2	9.06	400	610	1	420	88620
3	15.06	611	1000	1	309	120510
4	14.15	611	1000	310	656	135330
5	**17.38**	**400**	**610**	**421**	**574**	**32494**
6	14.41	1	399	346	688	136857
7	15.39	1	399	689	1000	124488
8	9.13	400	610	575	1000	89886
9	13.98	611	1000	657	1000	134160

Figure 4.26: The new distributions, the associated processor timings, partition range limits and workloads are shown for iteration 2.

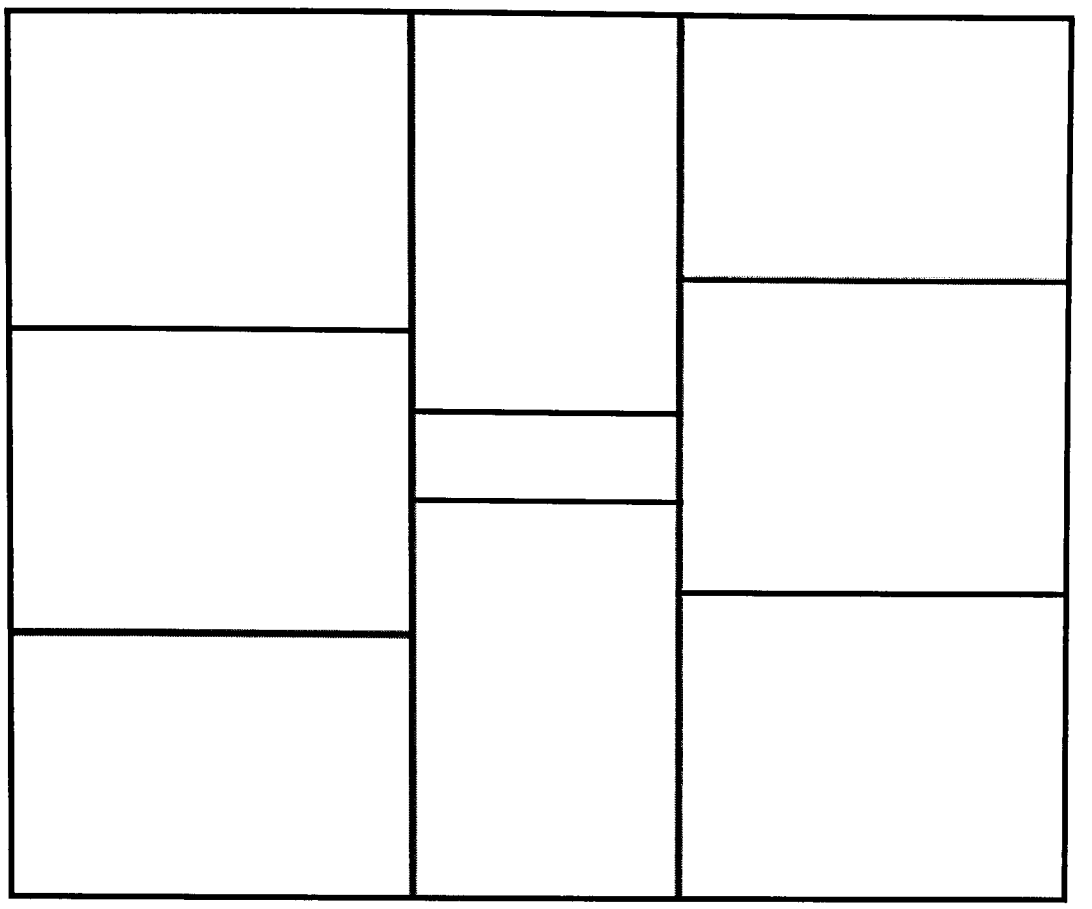
Iteration 3:



Processor	Timing	Left	Right	Up	Down	Workload
1	12.78	1	354	1	350	123900
2	11.63	355	610	1	438	112128
3	14.35	611	1000	1	295	115050
4	14.37	611	1000	296	647	137280
5	**16.53**	**355**	**610**	**439**	**557**	**30464**
6	12.99	1	354	351	701	124254
7	12.83	1	354	702	1000	105846
8	11.56	355	610	558	1000	113408
9	14.44	611	1000	648	1000	137670

Figure 4.27: The new distributions, the associated processor timings, partition range limits and workloads are shown for iteration 3.

Iteration 4:



Processor	Timing	Left	Right	Up	Down	Workload
1	13.57	1	372	1	352	130944
2	12.05	373	633	1	447	111667
3	13.57	634	1000	1	296	108632
4	13.59	634	1000	297	648	129184
5	**13.86**	**373**	**633**	**448**	**546**	**25839**
6	13.82	1	372	353	700	129456
7	13.77	1	372	701	1000	111600
8	12.14	373	633	547	1000	118494
9	13.55	634	1000	649	1000	129184

Figure 4.28: The new distributions, the associated processor timings, partition range limits and workloads are shown for iteration 4.

Iteration 16:

Processor	Timing	Left	Right	Up	Down	Workload
1	13.53	1	372	1	352	130944
2	12.07	373	633	1	447	111667
3	13.62	634	1000	1	296	108632
4	13.57	634	1000	297	648	129184
5	**13.93**	**373**	**633**	**448**	**546**	**25839**
6	13.45	1	372	353	700	129456
7	13.74	1	372	701	1000	111600
8	12.12	373	633	547	1000	118494
9	13.49	634	1000	649	1000	129184

Figure 4.29: The processor timings, partition range limits and workloads are shown for iteration 16.

It is evident that the workload on the middle processor is being reduced as expected. The initial workload on this processor was 333^2=110889 cells, where after the initial redistribution it has been reduced to 211x154=32494 cells, and then it is reduced to 256x119=30464 cells after the second redistribution. The workload on the middle processor after the third redistribution is just 261x99=25839 cells.

The serial (1x1) wallclock time taken to process iteration 16 of the modified JACOBI code is 122.13 seconds on a SunBlade 100 (Table 4.1), whereas the 3x3 wallclock times for the non-DLB and DLB parallel code (using the topology described above) are 64.80 and 17.27 seconds respectively. Therefore the speed up for the non-DLB execution is 1.88 as opposed to a speed up of 7.07 when DLB is used, highlighting the benefit of using DLB.

Iteration 16	Wallclock	Speed Up
1x1 (serial)	122.13	-
3x3 without DLB	64.80	1.88
3x3 with DLB	17.27	7.07

Table 4.1: Wallclock times and speed up for iteration 16 of the modified JACOBI code when using a 3x3 processor topology with and without DLB.

This test has demonstrated that the algorithm to determine the new processor workloads behaves as expected, shifting the load off the slow processor(s) onto the faster processors, whilst reducing the maximum processor timing. This test has also hinted towards the use of the ratio of the maximum and average processor timing as an indicator of when to redistribute. Numerous runs of the DLB and non-DLB parallel code suggest that there is no need to redistribute the workload when the ratio is less than 1.16 say. In some instances the timing of a processor would oscillate between iterations, and then return to its previous timing which was balanced with the other processors. Using the ratio prevented a redistribution occurring simply because of a temporary surge in processor usage and because the redistribution time was small, but ensured that redistribution did occur when the timings appeared imbalanced.

4.9.2 The APPLU-1.4 And ARC3D Codes

The JACOBI code was a simple code which did not exhibit all of the traits necessary for thoroughly testing the DLB strategy, and so natural progression led to the manual implementation of the DLB Staggered Limit Strategy within the APPLU-1.4 and ARC3D codes.

The APPLU-1.4 code is a self-validating 3323 line real world style CFD solver that is part of the NAS benchmark suite [88]. It was developed by the NASA Ames Research Center to evaluate the performance of parallel supercomputers. It does not perform an LU factorisation, but instead implements a symmetric successive over-relaxation (SSOR) numerical scheme to solve a regular-sparse, block lower and upper triangular system where most of the computation occurs in the routine SSOR that calls the routines BUTS and BLTS. As with the JACOBI code, this application is computationally balanced (i.e. it is physically balanced).

The ARC3D code is a real world application that uses an implicit Euler solver. Developed at the NASA Ames Research Center this physically balanced application used to be part of the PERFECT benchmark suite [89].

As well as having to deal with implicitly partitioned data (see Section B.7.1 and Figure 4.9 in Sections 4.3), these two applications tested the functionality of special and offset DLB communications. In terms of implementing the new distribution, both codes required the migration of several variables in each dimension, as well as the need for duplicating overlap communications. The experience gained from manually implementing the DLB strategy within these two applications was then applied to the SEA code (see next Section) and was also used in the development of algorithms for automating the implementation process (see Chapter 5).

4.9.3 The SEA Code

The Southampton-East Anglia Model (SEA code) [90] is a 7303 line code that uses an oceanography model to simulate the fluid flow of the ocean across the globe. Developed jointly by Southampton University and the University of East Anglia, a discretised model of the Earth (180x73x15 cells) is used with varying ocean depths in the third dimension. The CAPTools generated parallel code is partitioned evenly onto a number of processors, each of which may own a number of land cells and a number of ocean cells, as shown in the 3x3 processor topology in Figure 4.30.

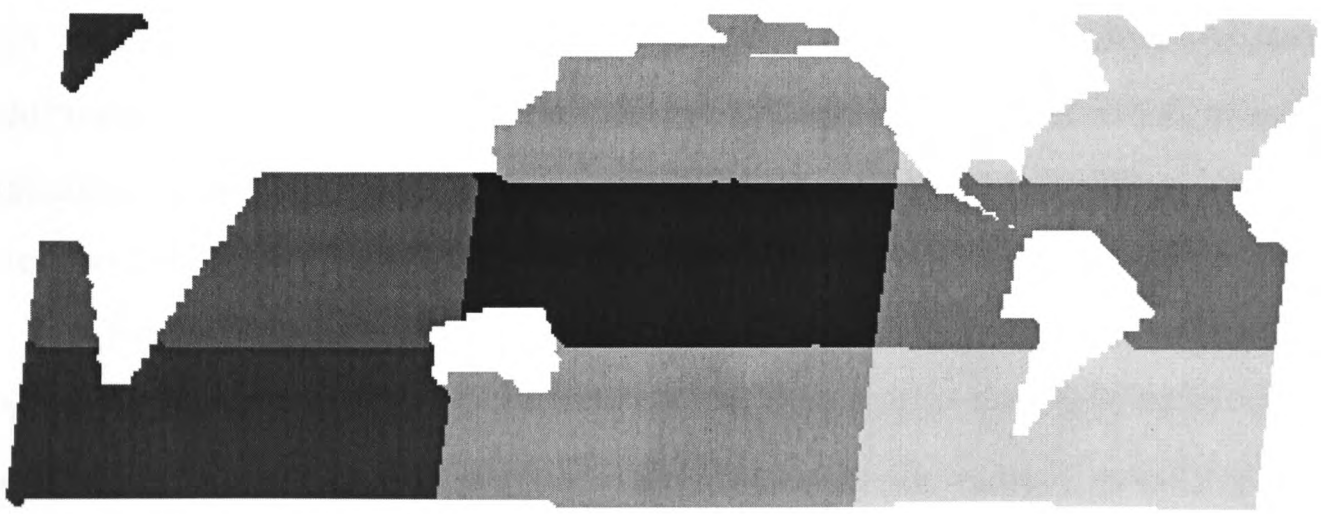


Figure 4.30: A discretised model of the Earth is evenly partitioned onto 3x3 processors (each represented by a different shading), where each processor owns a varying depth of ocean upon which to compute on.

The problem of parallel inefficiency arises naturally in the oceanography code. When trying to model the flow of the ocean in the fluid flow solver, few calculations are performed on processors owning a high proportion of land cells. This means that some processors will remain idle whilst waiting for other processors to complete their calculations, exhibiting natural imbalance since the amount of computation depends on the varying depth of ocean. For example, there are many more fluid flow computations in the Pacific than there are in the North Atlantic, and so if the load is evenly mapped then the processor containing Europe and Russia (in the top left corner) would be idle whilst waiting for the other processors containing ocean cells to finish computing.

When running this code on a homogeneous parallel machine such as the CRAY T3E [116], where the imbalance is predominantly due to physical characteristics, it becomes apparent that there is a need to balance this type of problem differently from when the imbalance is due to the variation between processors. For example, if a processor owning mainly land cells were to gain chiefly ocean cells at its own weight then it would assume that it would be gaining more land cells. This is not the case, and it would in fact gain far too many ocean cells thinking that it could process them quite quickly. What should happen in this instance is that this processor should gain ocean cells that are of a different weight, which should reduce the amount of ocean cells that it would take on, as ocean cells are processed at a slower rate. The only reason it had a small weight was because it had very little work to do, and not that it was a fast processor, therefore the processor should take on the weight of the cell and not assume that it can process these extra cells at its own rate. This highlights the point that additional cells should be processed at the processor weight when processor imbalance is assumed, and at the cell weight when physical imbalance is assumed (Section 1.11).

Figure 4.31 shows the processor timings for a single iteration (Iteration 16) using the different balancing techniques. A snapshot of Iteration 16 was given because the code tended to perform a lot of computation in the first two iterations (due to the initial conditions), and so using a snapshot at some later iteration was deemed a fairer comparison as the load balancing strategies had a chance to stabilise. The timings shown are for the unbalanced code, the code balanced using global processor partition range limit changes, and for the code balanced using the

non-coincidental processor partition range limits assuming processor and physical imbalance. The processor timings appear imbalanced when no load balancing is undertaken, which suggests that there is a fair amount of idle time present in the system. There is very little work being done by the processor that owns Europe and Russia (Processor 9 in this case, in which a 4x3 processor topology is being used). The maximum processor time can be reduced simply by balancing the workload on each processor using the given methods, but the best result in both overall time and load balance is achieved when staggering the limits assuming physical imbalance (where processor 9 is then given a sufficient amount of work).

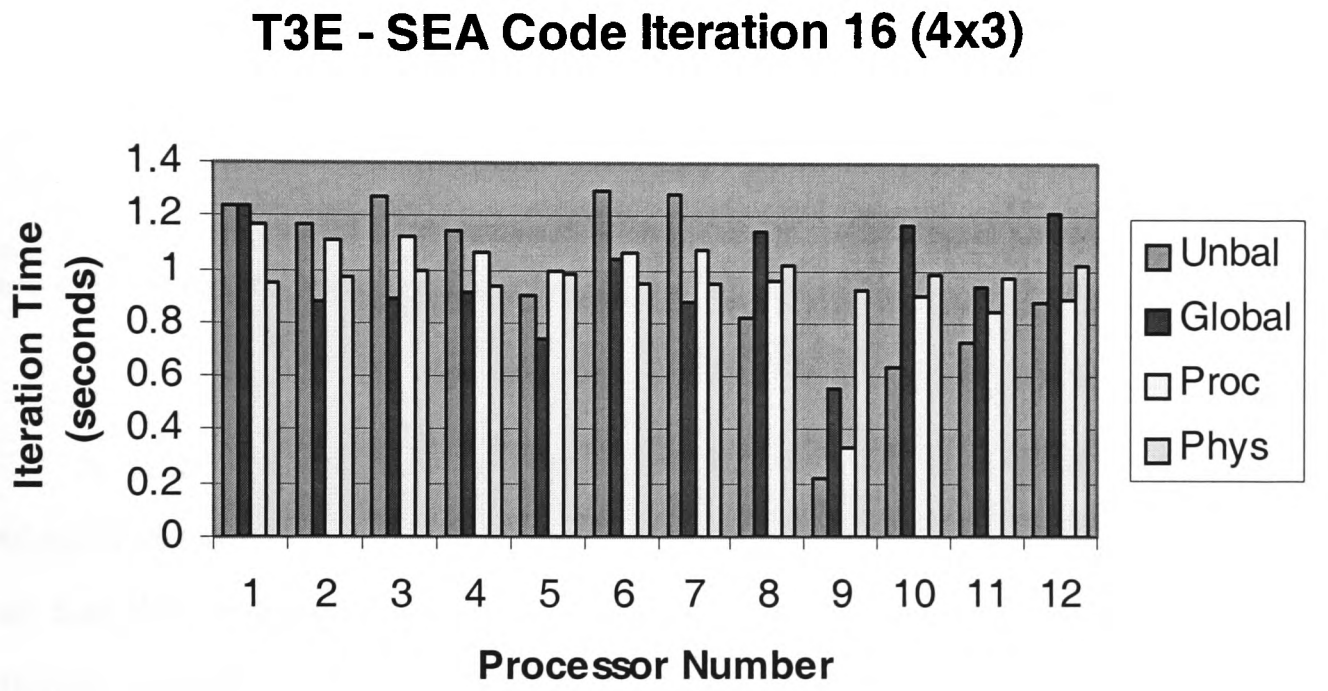


Figure 4.31: Processor timings at Iteration 16 for various types of load balancing techniques, where Processor 9 contains Europe and Russia.

A more general overview can be seen in Figure 4.32, in which statistical measurements are given for each of the different balancing techniques. The aim is to reduce the maximum iteration time down towards the average time, from which it is apparent that this is best achieved when balancing the problem assuming the correct imbalance type. The load is not sufficiently balanced when changing the limits globally, and the load is overestimated when assuming that there is processor imbalance. When a light processor gains cells from a heavy processor then it gains too many cells when assuming processor imbalance, because it thinks that it can process those extra cells quickly at its own rate. The load is correctly balanced when assuming physical imbalance, in which each processor has the

same speed but a differing workload, where it can be seen that there is less idle time (making more efficient use of the available hardware).

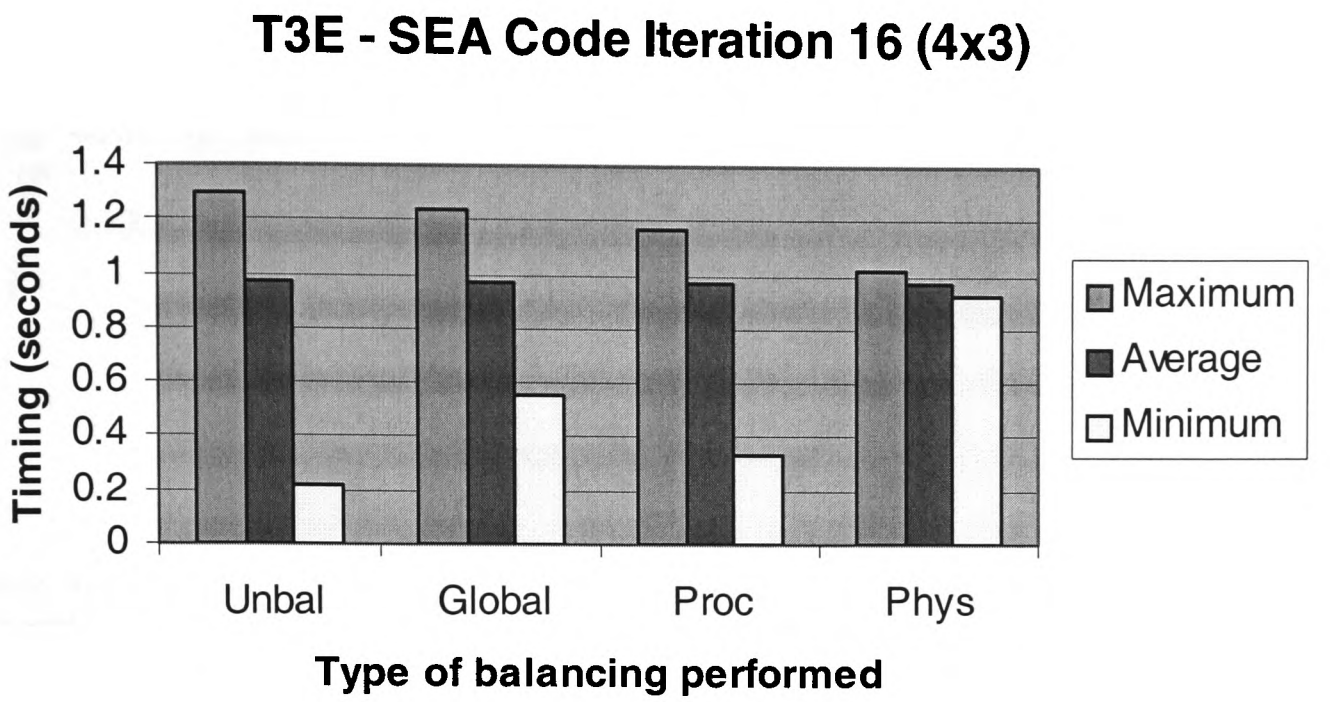


Figure 4.32: Statistical measurements for the various load balancing techniques at Iteration 16.

A similar trend can be seen for the various homogeneous processor topologies shown in Figure 4.33, in which any form of balancing is better than none, and that staggering the limits is better than changing them globally. In this particular instance it is better to assume physical rather than processor imbalance, which implies that the type of imbalance being addressed needs to be considered when performing DLB.

Additionally, since these times include the redistribution time, it can be seen that the algorithm is cheap enough to be used and so significant speed improvements can still be achieved.

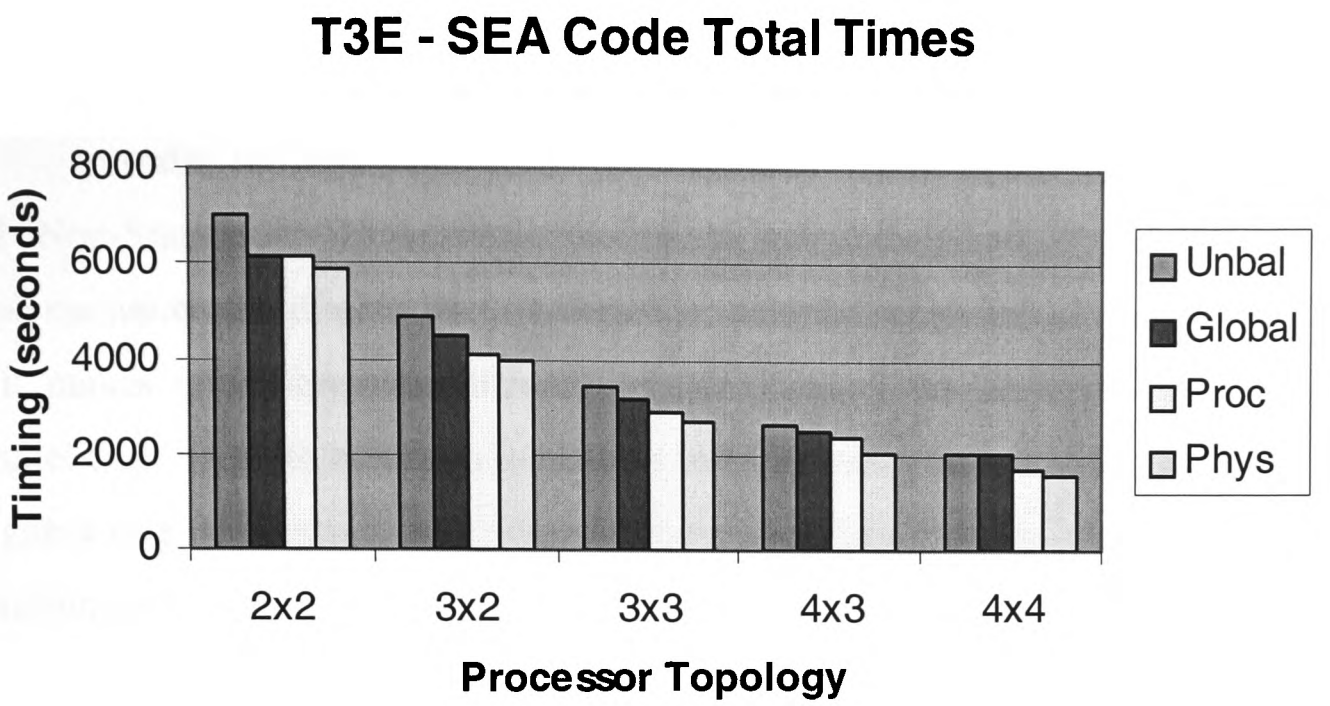


Figure 4.33: The execution times (CPU+Redistribution time) for 2000 Iterations using different load balancing techniques on various processor topologies.

The DLB strategy appears relatively effective on the SEA code even when the number of processors is increased. Had this code been performing calculations just on land cells, then the level of load imbalance would have been far greater than it is here as many of the processors would have had little or no calculations, and so it is suspected that the effectiveness of the DLB strategy possibly would have been far greater.

4.10 Summary

This Chapter has discussed the manual implementation of the DLB Staggered Limit Strategy within an existing CAPTools generated parallel code. Using a parallel code that already exists reduces the amount of effort needed to produce a DLB parallel code, highlighting the benefit of using CAPTools as a starting point, since it lays the foundations upon which to work. If CAPTools were not used in this research, then the program of work would have to include the parallelisation of codes as well as the implementation of dynamic load balancing. Implementing the DLB strategy within an already existing parallel code also allows for the

comparison between the DLB and non-DLB parallel versions, which may be more problematic if starting from scratch.

In order to implement the DLB Staggered Limit Strategy, the Staggered and Non-Staggered Dimensions needed to be determined. With a manual implementation the dimension that should contain the staggered limits is arbitrary, as it makes no difference which dimension contains the staggered limits. The parallel code needs to be set up to execute in DLB mode, which means identifying neighbouring processors and their processor partition range limits. Existing communications can then be identified and converted into DLB communications, where only those communications in a Non-Staggered Dimension involving the staggered limits need to be converted.

Information from a code execution profile, or knowledge from a user, could be used to determine where to redistribute the workload, where calls to specific DLB utilities are placed in the necessary location. In order to implement the DLB strategy the workload needs to be migrated using dedicated migration calls. Migration calls are constructed for the data in each dimension separately, (i.e. grouping them according to the migrated dimension) and the processor partition range limits are updated before migrating in a subsequent dimension. After migrating the workload, each processor owns the data defined by its new limits but they also need to know the values in their halo region in order to continue executing. The processor partition range limits are updated internally and certain duplicated halo communications are then executed.

This Chapter has shown that it is possible to successfully implement the DLB Staggered Limit Strategy within an existing parallel code and it has also highlighted the difficulties surrounding manual implementation. Results in the JACOBI code and SEA code have shown improvements in parallel performance due to employing this strategy. With the number of alterations required there is much scope for introducing errors including incorrect communication conversion, incorrect construction of migration calls and incorrect communication duplication. These potential pitfalls all prolong the implementation time since debugging would be required. An added difficulty with larger codes is that it may be difficult to identify the necessary communications for duplication, which then need to be tested to determine whether the usage occurs after redistribution.

Chapter 5 Automatically Implementing The DLB Staggered Limit Strategy Within CAPTools Generated Structured Mesh Codes

The previous Chapters have discussed the necessity for DLB, and have discussed how to manually implement the DLB Staggered Limit Strategy into a parallel code using the newly developed generic utilities. This Chapter examines how to automate the process of implementing the DLB Staggered Limit Strategy within a CAPTools generated parallel code, following the manually implemented techniques (Chapter 4).

5.1 Automation Within CAPTools

Implementing any DLB strategy within a parallel code (or even more so from scratch within a serial code) can be a tedious and time-consuming process. For this reason it is desirable to automate the whole process so that less time is spent on the mundane task of implementation, enabling more time to be spent on testing and obtaining results. The bulk of the effort required to implement the DLB Staggered Limit Strategy involves enabling processors to communicate across the staggered limits, and also to ensure that the correct transfer of data between processors satisfies any new partition (and the halo region of that partition). In effect this entails identifying and converting particular communications generated in the Non-Staggered Dimensions into DLB communications, identifying and constructing the necessary migration calls, and duplicating any necessary overlap communications.

Many DLB strategies have been implemented within specific applications [78, 79, 91, 92, 93, 94, 95, 96, 97], where the developer has expert knowledge of the code. For example, Burton et al. [98] investigate and implement numerous load balancing strategies in the UK Met. Office's Unified Model. They suggest that the techniques that they describe can be applied to other application codes

with similar characteristics, which is also true for the DLB Staggered Limit Strategy. Automation makes it possible for a non-expert user of the application code to generate a DLB parallel version of the code, where the time to implement DLB is reduced dramatically to seconds/minutes.

One of the main reasons why the DLB Staggered Limit Strategy can be automated is that the actual algorithm was devised to be generic and so it could be applied to a wide range of application codes. This makes it a suitable feature to include within CAPTools, since CAPTools aims to be applied to a wide range of real world applications (Section 1.8).

The manual implementation of the DLB Staggered Limit Strategy is possible within a CAPTools generated parallel code using various communication transformations and by inserting some new DLB code (Chapter 4). During manual implementation it was found that the same operations were being performed numerous times, indicating that this was a definite candidate for automation.

The CAPTools generated parallel code is created internally before it is generated (Section B.12), and so it is possible to internally transform this into a CAPTools generated DLB parallel code. Using existing data structures within CAPTools, algorithms can be constructed to automatically implement DLB within a variety of structured mesh application codes. For example, a single procedure can be used to generate each of the different migration calls, and similarly a single procedure can be used to convert existing communications into DLB communications. Additionally, the way in which the DLB implementation code is set up is the same for each application code (Section 5.7).

5.2 Adding DLB To The Functionality Of CAPTools

Naturally, a new feature of CAPTools would have to be installed as part of the graphical user interface, extending its functionality to include this new DLB option. Note that the current functionality of CAPTools should still be retained and so the selection of the DLB option should only be a choice and not a requirement.

The user should be able to generate a DLB parallel version of their serial code easily using CAPTools. Unlike the manual implementation of the DLB strategy, where an already generated CAPTools parallel code was transformed into a DLB parallel code, with automation it is now possible to implement the DLB strategy at any stage during the parallelisation process (Figure B.1).

If the DLB option were selected at the beginning of the parallelisation process (see Figure 5.1a) then this would be acceptable if every generated parallel code were to include the DLB implementation. If the current functionality of CAPTools is to be retained then CAPTools should still be able to generate parallel codes without DLB (i.e. non-DLB parallel codes). This means that if a non-DLB parallel code needs to be generated, having already generated a DLB parallel code, then the whole parallelisation process would have to be repeated, this time without selecting the DLB option. An additional 'De-Implement DLB?' option could be provided to convert the DLB parallel code into a non-DLB parallel code, but this option would require more effort in terms of implementing this approach within CAPTools and would also deviate from the parallelisation process already in use.

The DLB option could be provided during an iteration of the parallelisation process before partitioning another dimension (i.e. after generating communications). For example, after partitioning an application code in dimension 1 followed by dimension 2, the user could decide to select the DLB option and then go on to partition dimension 3. As mentioned above, this would lead to difficulties in producing a non-DLB parallel code, in the sense that a 'De-Implement DLB?' option would be required. In particular, enabling the DLB option to be selected in this manner would essentially fix the Staggered Dimension to the current partitioned dimension in which the option was selected, which introduces additional problems (see Section 5.3).

Ideally, the DLB option should not affect any stage of the parallelisation process, and so the generation of DLB parallel code should be provided at the end of the current parallelisation process (see Figure 5.1b). In this way, all existing CAPTools algorithms remain the same. It is then possible to generate both a non-DLB and a DLB parallel version of the serial application code (after the communication phase) without having to repeat the parallelisation process. The

communications database can simply be loaded into CAPTools and the required DLB option selected.

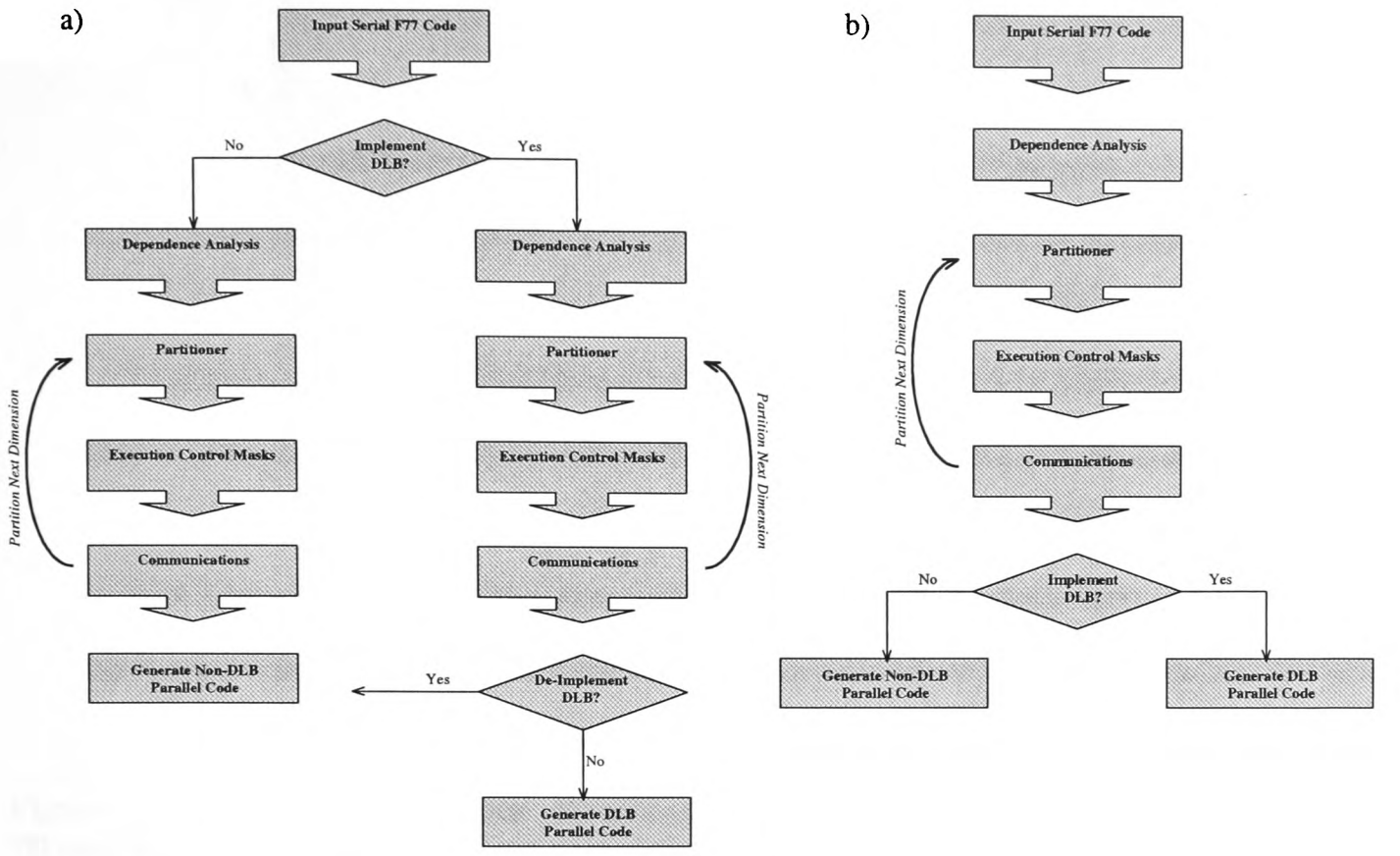


Figure 5.1: Pictorial representation of the parallelisation process when the user is given the option to implement DLB a) from the onset, or b) at the end of the parallelisation process.

The Code Generator window (Figure B.46) has been modified to enable the user to select the DLB option at the end of the parallelisation process, and is shown in Figure 5.2. If the user decides not to partition another dimension, then a non-DLB parallel version of the serial application code can be generated using the usual "Generate & Save Final Code" button. Alternatively, the "Dynamic Load Balance" button can be used before generating and saving the final parallel version of the serial application code in which DLB has been implemented within.

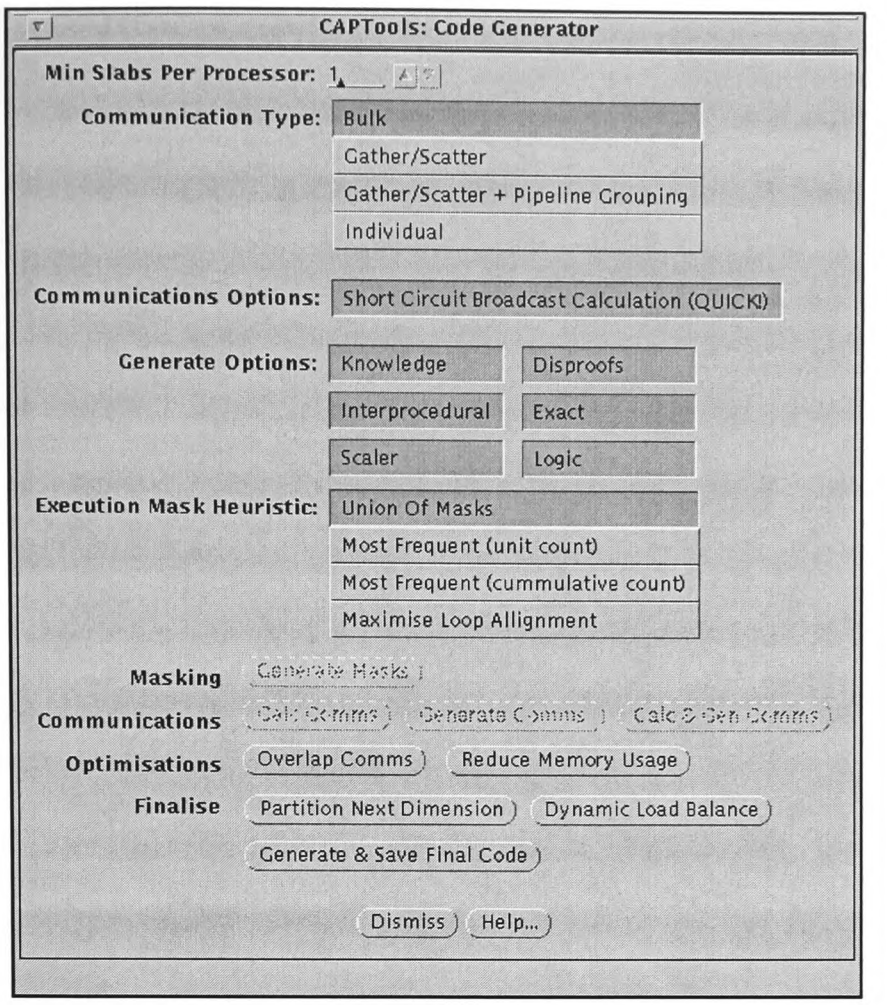


Figure 5.2: The Code Generator window (see Figure B.46) is modified to include the "Dynamic Load Balance" button as part of the functionality of CAPTools.

At present the task of identifying load imbalance within an application code is left to the user using a code execution profiler or knowledge of the code. If the user decides to select the "Dynamic Load Balance" option in Figure 5.2, then the new self-contained DLB Browser window is displayed, enabling the selection of the loop containing a significant amount of load imbalance (Figure 5.3). The user is presented with a list of all the routines in the application code, for which all possible loops are displayed upon selection of a particular routine. Once the user is content with their choice of DLB Routine and DLB Loop containing the load imbalance then the selection of the Apply button will implement the DLB strategy within their parallel code. The user can then generate and save the final code after implementation using the option available in the modified Code Generator window.

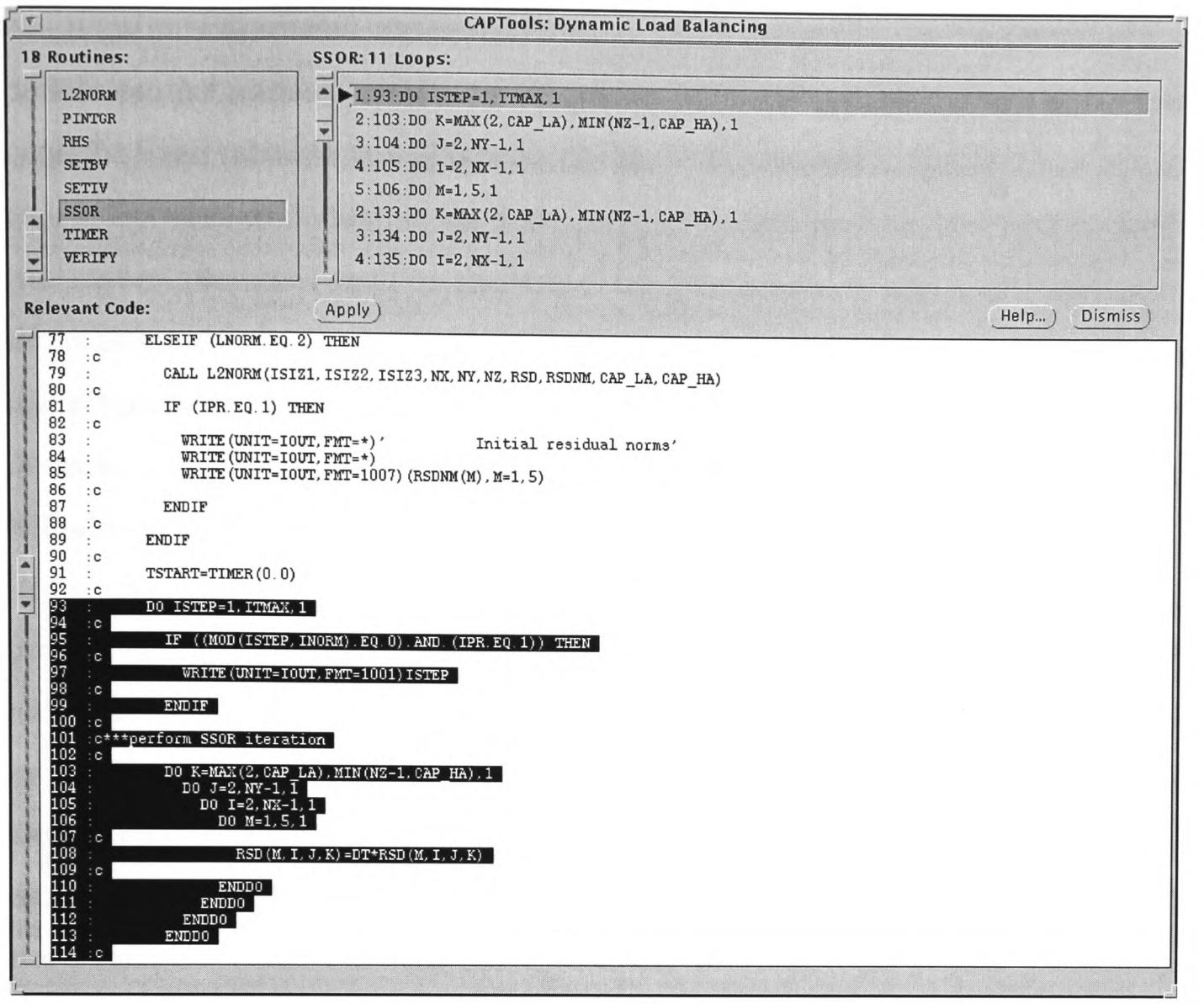


Figure 5.3: The DLB Browser window used to select the imbalanced loop.

5.3 Fixing The Staggered Dimension

Having decided that the DLB option will only be available to the user in the final stages of the parallelisation process, the final issue is to determine which dimension should contain the non-coincidental (staggered) limits. With the manual implementation of the DLB strategy the dimension containing the staggered limits was arbitrarily chosen, where all of the necessary partitioning details were obtained by examining the code. However, all of the necessary information is not available when automating this process, since CAPTools tries to store the minimal amount of data possible, only storing information pertaining to the current partition dimension. Therefore this issue is taken into account when deciding which dimension shall contain the staggered limits once the DLB option is selected.

With automation the user could be given the option to either select the dimension that should contain the staggered limits, or the Staggered Dimension could be fixed such that it will always be the n^{th} dimension for instance.

For ease of coding, it has been decided that the last partitioned dimension (the current partition) shall be the Staggered Dimension containing the staggered limits as information contained in the data structures of the current partition aides the automation process. The user needs to be aware that the last partitioned dimension shall contain the staggered limits, so they should partition their data with this in mind.

Had the option to implement DLB been provided during the parallelisation process (and not at the start or just before generating the final code, as discussed in Section 5.2), where the Staggered Dimension was set to the partition dimension in which the DLB option was activated, then information relating to the Staggered Dimension would need to be stored internally within CAPTools. In addition, the automation algorithm for this approach would involve unnecessary complications. For example, if the user decided to activate the DLB option after generating communications for the third partitioned dimension, then communications in the first and second dimension would need to be converted into DLB communications. Any communications generated in any further partitioned dimensions would also need to be converted into DLB communications (or generated directly without conversion). Additionally, the DLB implementation algorithm would not be independent (where the entire procedure can be executed in one stage), involving more effort when maintaining the algorithm within CAPTools. As mentioned in Section 5.2, all of these transformations would need to be removed if a non-DLB parallel code needed to be generated. If the user activates the DLB option in the last partitioned dimension, then this would mean converting existing communications in previously partitioned dimensions using the current partition information, allowing the DLB algorithms to be independent from existing CAPTools algorithms.

5.4 New Data Structures Needed For Automation

At present CAPTools only stores information associated with the current partition (in the Staggered Dimension), however some information relating to all partitions needs to be known when generating migration calls since the stride and the processor partition range limits need to be known for each partitioned dimension (Section 5.8).

When the user decides to partition another dimension, instead of CAPTools automatically deleting the information relating to the previous partition (Section B.11), a new field is set up to store the partition details (Figure 5.4) so that the current partition, along with all previous partitions, can be stored for each routine.

```
POLDPARTITIONLIST=^OLDPARTITIONLIST;
OLDPARTITIONLIST=RECORD
  PARTITION:PPARTITION;
  NEXT:POLDPARTITIONLIST;
  END;

PROUTINE=^ROUTINE;
ROUTINE=RECORD
  PARTITION:PPARTITION;
  OLDPARTITIONLIST:POLDPARTITIONLIST;
  ....;
  NEXT:PROUTINE;
  END;
```

Figure 5.4: New data structure needed to store information relating to the current and previous partitions of a particular routine.

5.5 Overview Of Automatically Implementing The DLB Staggered Limit Strategy

There are two major components involved in automatically generating a DLB parallel code (Figure 5.5). The first ensures that the parallel application still operates correctly when the staggered limits are employed, and consists of identifying and changing existing communications that need to be converted into DLB communications (Sections 3.3 and 4.3). After the initial redistribution, the

processors need to be able to communicate across the staggered limits to non-immediate neighbours, which can be achieved using the DLB communications.

The second component ensures the correctness of the parallel code after redistribution (Section 1.14.4), and involves migrating data between processors to conform to the newly calculated processor partition range limits. This component determines for each partitioned dimension what data needs to be migrated in that particular dimension and constructs the necessary migration call for every array affected by the altered limits. The processor partition range limits for a particular dimension are then updated after constructing all of the required migration calls. Finally, this component identifies and duplicates communications that update the overlap region, where the communicated overlap region is assigned before redistribution and is used after redistribution.

- *Identify and convert existing communications in Non-Staggered Dimension into DLB communications*
- *Insert DLB implementation code*
  - *Initialise DLB mode and add necessary code to enable dynamic load balancing*
  - *Add migration calls for each dimension*
  - *Duplicate necessary overlap communications*

Figure 5.5: The major components involved in automatically generating DLB parallel code using CAPTools.

5.6 Identifying And Converting Existing Communications Into DLB Communications

Due to the use of staggered limits, processors may potentially have to communicate with several neighbours in a Non-Staggered Dimension, meaning that the original communication message will need to be dissected. This means that communications orthogonal to the Staggered Dimension will need to be converted into DLB communications if they are also partitioned in the Staggered Dimension (Section 3.3). The communication call name is changed to reflect the DLB status, and four extra parameters (FIRST, STAG_STRIDE, LOWLIM, and HIGHLIM) are added to the parameter list as discussed in Section 4.3. To

automate this whole process, those communications that need to be converted into DLB communications must first be identified and information relating to the current partition can be used to set up the additional parameters (Sections 5.6.1 and 5.6.2).

5.6.1 Identifying Those Communications To Be Converted

The first stage is to identify those potential communications throughout the code that may need to be converted, which means examining all communications generated in previously partitioned dimensions. The second stage is to determine whether the communicated data is also partitioned in the current pass (i.e. partitioned in the Staggered Dimension), since only those communications that are affected by the staggered limits need to be converted. If the communicated data is not also partitioned in the Staggered Dimension then there is no need to convert the communication into a DLB communication.

Every statement in the parallel code is tested to see if it is a communication call, which involves processing every command in every block of an active routine, and so the routines are processed in their STRICT order (Section B.3). Any intrinsic Fortran functions or any CAPLib function (such as CAP_INIT for instance) do not need to be processed since these will definitely not contain any communications. Communications generated in the current pass can be ignored, as these communications will not need to be converted since processors shall always be communicating with their immediate neighbour in the Staggered Dimension.

After identifying that the statement is a CALL statement, the RECEIVE data structure (Section B.9.2) for the statement being processed can be used to identify if this is a communication call. If CCOMMAND^.RECEIVE is not NIL then this indicates that the statement is a communication statement that was generated in the current partition (which is of no interest), whereas a NIL value indicates that the communication was generated in a previously partitioned dimension (i.e. in a Non-Staggered Dimension). Note that only certain types of communication calls need to be converted into DLB communications, where for

example Broadcasts will not need to be converted since the communicated data will be broadcast to all other processors.

Having identified a communication statement generated in a Non-Staggered Dimension, the communicated data of that statement must be examined. If the communicated data is also partitioned in the Staggered Dimension (the current partition) then this orthogonal communication will certainly be affected by the staggered limits, meaning it needs to be converted into a DLB communication. If the communicated data is partitioned in the Staggered Dimension then it will be found in the current partition list for the routine in which the data is communicated, CROUTINE^.PARTITION (Section B.7.2). If the communicated data is not found in the current partition list (i.e. is unpartitioned) but found to have an execution control mask on its assignments in the current partition then it may be treated as if it was partitioned (Section B.9.1.2), otherwise this communication need not be converted into a DLB communication. Figure 5.6 shows the basic pseudo code algorithm used to identify those communications that will need to be converted into DLB communications.

```
For every command in every routine
If command is a communication statement generated in a previous pass then
  See if communicated data is partitioned in the Staggered Dimension
  If not partitioned then
    See if communicated data is implicitly partitioned via currently masked
    assignment statements
  End If
  If communicated data is partitioned (in some way) then
    Convert communication into a DLB communication
  End If
End If
```

Figure 5.6: The basic pseudo algorithm used to identify those communications that may need to be converted into DLB communications.

5.6.2 Converting Communications Into DLB Communications

The communication name needs to be changed to reflect that this communication is now capable of exchanging information across staggered limits. In addition,

four parameters need to be added onto the parameter list. Figure 5.7 shows an example communication call that has been converted into a DLB communication along with the converted tree structure (where the changes to both are shown in bold), and this can be compared to the tree structure of the original CAP_SEND communication shown in Figure B.64.

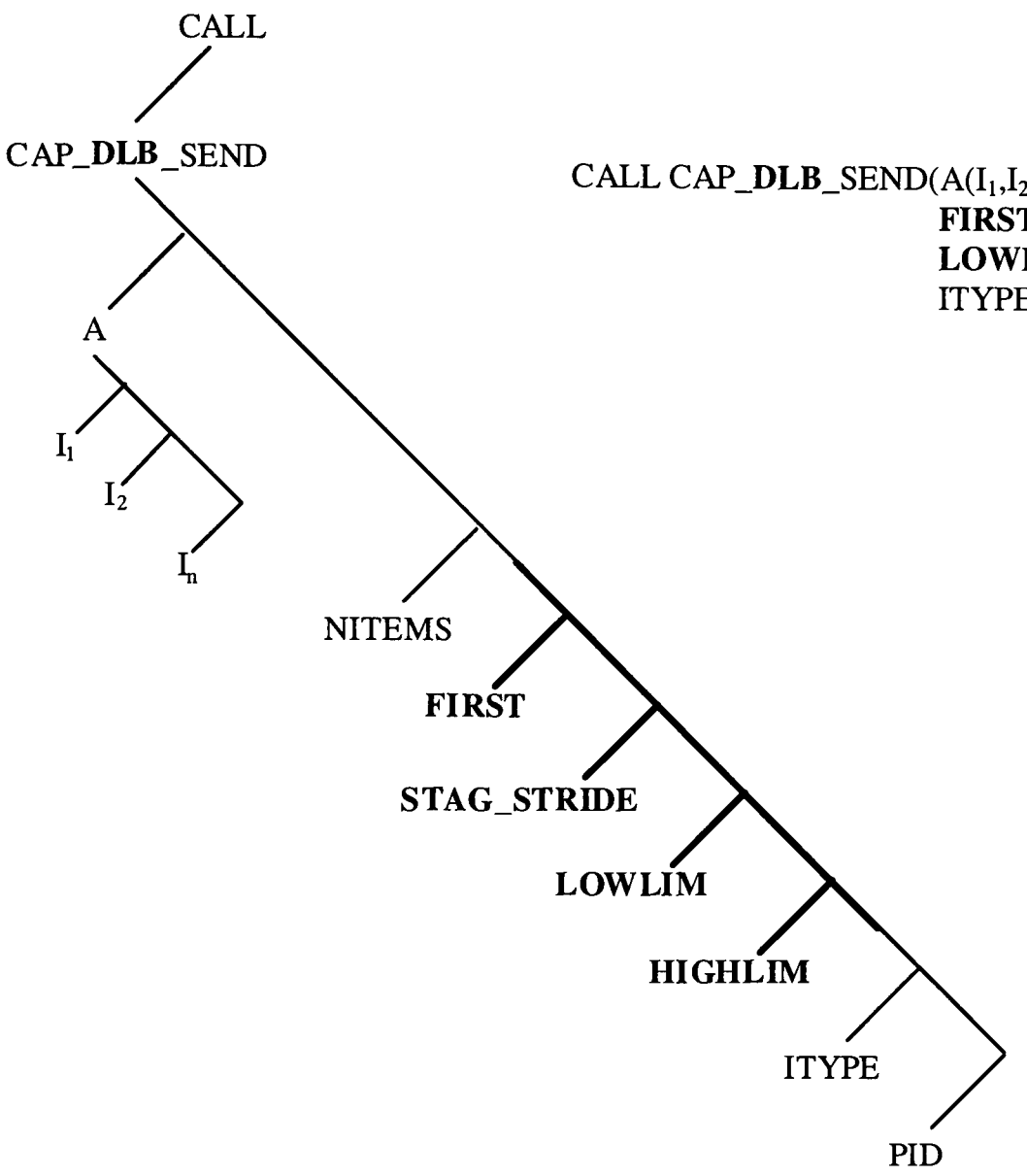


Figure 5.7: Example communication call (CAP_BSEND) that has been converted into a DLB communication call, where its associated tree structure is also shown.

The name of the communication call needs to be modified so that the call can be distinguished from non-DLB communications, which can be achieved simply by renaming the call, as illustrated in Figure 5.8. However, the converted call needs to retain its communication status so that for example CAP_DLB_SEND is still considered to be a CAP_SEND communication call having a KIND of KEYSEND. This is necessary since the information is used in other DLB procedures. The call to MATCHREFERENCE ensures that the called communication is linked to the routine in which it is called, where the correct call

graph is set up. The modified communication type is then set to that of the original communication.

```
(* Store the communication kind *)
COMKIND:=CCOMMAND^.LINK^.LEFT^.SYMB^.KIND;
IF (COMPSTRING(CCOMMAND^.LINK^.LEFT^.SYMB,'CAP_SEND ',9)) THEN
   SETHAS(CCOMMAND^.LINK^.LEFT^.SYMB,'CAP_DLB_SEND')
ELSEIF ...
...
( Adjust call graph *)
MATCHREFERENCE(CCOMMAND^.LINK^.LEFT,CCOMMAND^.LINK^.LEFT^.SYMB^.NAME,
                        CROUTINE,TOPOFFILE,CCOMMAND,ERROR);
(* Ensure modified communication is still of the same kind *)
CCOMMAND^.LINK^.LEFT^.SYMB^.KIND:=COMKIND;
```

Figure 5.8: Code used to convert a communication call name into a DLB call, where the type of communication is retained.

The four additional DLB parameters (FIRST, STAG_STRIDE, LOWLIM and HIGHLIM) need to be included in the call parameter list, which means extending the communication tree by creating a new branch for each extra parameter. These parameters are located at the same place within the call list for all types of DLB communications, before ITYPE (which is always the second-to-last parameter in the call). Figure 5.9 illustrates how to identify the location at which to place these new parameters.

```
(* Identify the parameter before ITYPE in the communication statement *)
TREE:=CCOMMAND^.LINK^.LEFT;
(* Traverse to the third but last parameter *)
WHILE (TREE^.RIGHT^.RIGHT^.RIGHT <> NIL ) DO
   TREE:=TREE^.RIGHT;
```

Figure 5.9: Code used to identify the location in the communication tree structure at which to place the additional DLB parameters.

The additional DLB parameters can be obtained using the current partition information for the routine in which the communication is contained, shown in Figure 5.10. The starting index value (FIRST) of the communicated data in the Staggered Dimension (Section 3.3.1.1), can be extracted from the starting address of the communication using PARTITION^.INDEX, which stores the currently partitioned index (i.e. the staggered index). This index is then used to traverse the tree of the communication starting address, as shown in Figure 5.11. For example,

if the current partition index is 2 then FIRST will need to be set to the value of the second index in the starting address. The tree of FIRST is then linked into the tree structure of the communication call.

```
PARTITION^.SYMBOL
PARTITION^.INDEX
PARTITION^.MODDIVOFFPTR^.MODNONLOOP
PARTITION^.MODDIVOFFPTR^.MODCONST
PARTITION^.MODDIVOFFPTR^.DIVNONLOOP
PARTITION^.MODDIVOFFPTR^.DIVCONST
PARTITION^.MINSYMB
PARTITION^.MAXSYMB
```

Figure 5.10: The main fields of the PARTITION data structure in a given routine that are used to automatically convert a given communication into a DLB communication.

```
INDEX:=PARTITION^.INDEX;
(* Start looking at the starting address of the communication *)
CTREE:=CCOMMAND^.LINK^.LEFT^.RIGHT^.LEFT;
NEWTREE(DLB_FIRST_TREE);
IF (INDEX > 0) THEN
   BEGIN
   (* Get to the partitioned index of the variable being communicated *)
   FOR I=1 TO INDEX DO
      CTREE:=CTREE^.RIGHT;
   (* Set FIRST to the partitioned index *)
   DLB_FIRST_TREE^.LEFT:=TREECOPY(CTREE^.LEFT);
   END;
DLB_FIRST_TREE^.RIGHT:=TREE^.RIGHT;
TREE^.RIGHT:=DLB_FIRST_TREE;
TREE:=DLB_FIRST_TREE;
```

Figure 5.11: Code used to traverse to the partitioned index in the communication starting address (where the partition INDEX > 0).

If the communicated data is 1D-mapped, then the PARTITION^.INDEX will not have a positive value and FIRST will have to be extracted from the starting address expression in a different manner to that discussed above (see Section B.7.1). The staggered partition component can be extracted using EXTRACTEXPRESSION that uses the SYMBOLICMOD and SYMBOLICDIV (Table B.3) by first obtaining the remainder of the 1D expression when factorised by the Mod value, and then finding the factor of this result when using the Div value. The example shown in Figure 5.12 illustrates how to extract the partitioned component using EXTRACTEXPRESSION.

REAL T(l_1:h_1,l_2:h_2,l_3:h_3) –
INDEX 3 partitioned first, and INDEX 2 partitioned last
INDEX 2: Mod=(h_1-l_1+1)*(h_2-l_2+1), Div=(h_1-l_1+1)

```
SUBROUTINE SUB1(T)
REAL T(*)
...
CALL CAP_RECEIVE(T(1+((CAP2_LOW-1)-1)*(h1-l1+1)+
                                ((CAP1_LOW-1)*(h1-l1+1)*(h2-l2+1))),.....)
```

Remainder term of SYMBOLICMOD:
MOD((1+((CAP2_LOW-1)-1)*(h_1-l_1+1)+((CAP_LOW-1)*(h_1-l_1+1)*(h_2-l_2+1))),
(h_1-l_1+1)*(h_2-l_2+1))=1+((CAP2_LOW-1)-1)*(h_1-l_1+1)

Linearised factor term of SYMBOLICDIV:
DIV(1+((CAP2_LOW-1)-1)*(h_1-l_1+1),(h_1-l_1+1))=CAP2_LOW-1

Figure 5.12: When communicated data is 1D-mapped (i.e. INDEX ≤ 0), the partitioned component in the communication starting address for the Staggered Dimension can be extracted using EXTRACTEXPRESSION (which uses SYMBOLICMOD and SYMBOLICDIV).

Note that if the extraction fails (no factor is found), then CAPTools explicitly generates MOD and/or DIV functions in the application code.

The remaining DLB parameters (STAG_STRIDE, LOWLIM and HIGHLIM) can be extracted directly from the PARTITION data structure, as demonstrated in Figure 5.13. The stride in the Staggered Dimension of the communicated data can be set to the DIV component of the MODDIVOFFPTR field in the PARTITION record using BUILDTREE (Table B.3) to construct the parse tree from the nonloop expression. The new trees for LOWLIM and HIGHLIM (the DLB communication offsets) are usually equivalent to the MINSYMB and MAXSYMB fields of the PARTITION data structure, which are essentially the lower and upper processor partition range limits respectively. It may be the case that offsets are included in the DLB communication (see Section 3.3.1.2), meaning that LOWLIM and HIGHLIM will contain an expression rather than just the processor partition range limit.

```
(* Set up STAG_STRIDE *)
NEWTREE(DLB_STAG_STRIDE_TREE);
DIVNONLOOP:=PARTLIST^.MODDIVOFFPTR^.DIVNONLOOP;
DIVCONST:=PARTLIST^.MODDIVOFFPTR^.DIVCONST;
BUILDTREE(DIVNONLOOP,DIVCONST,CROUTINE,NIL,CCOMMAND,NIL,
                DLB_STAG_STRIDE_TREE^.LEFT,FALSE,TRUE);
DLB_STAG_STRIDE_TREE^.RIGHT:=TREE^.RIGHT;
TREE^.RIGHT:=DLB_STAG_STRIDE_TREE;
TREE:=DLB_STAG_STRIDE_TREE;


(* Set up LOWLIM *)
NEWTREE(DLB_LOWLIM_TREE);
NEWTREE(DLB_LOWLIM_TREE^.LEFT);
DLB_LOWLIM_TREE^.LEFT^.SYMB:=PARTLIST^.MINSYMB;
DLB_LOWLIM_TREE^.RIGHT:=TREE^.RIGHT;
TREE^.RIGHT:=DLB_LOWLIM_TREE;
TREE:=DLB_LOWLIM_TREE;

(* Set up HIGHLIM *)
NEWTREE(DLB_HIGHLIM_TREE);
NEWTREE(DLB_HIGHLIM_TREE^.LEFT);
DLB_HIGHLIM_TREE^.LEFT^.SYMB:=PARTLIST^.MAXSYMB;
DLB_HIGHLIM_TREE^.RIGHT:=TREE^.RIGHT;
TREE^.RIGHT:=DLB_HIGHLIM_TREE;
TREE:=DLB_HIGHLIM_TREE;
```

Figure 5.13: The STAG_STRIDE, LOWLIM and HIGHLIM parameters can be set up using the fields in the PARTITION record of the routine in which the communication is contained.

5.6.3 Implicit Partitioning Of Communicated Data And 'Special' DLB Communications

If the communicated data is not found in the PARTITION list, then this does not necessarily mean that the data is not partitioned and can be ignored. The communicated data may be implicitly partitioned (Section B.9.1.2), in which case the pseudo partition will need to be found in some other manner before converting this communication into a DLB communication, since unpartitioned data may also be affected by DLB. If the assignment of the communicated data is always masked in the same way, then it is possible to identify where that data is owned, enabling it to be treated as if partitioned.

When manually converting communications, even if the communicated data was implicitly partitioned, it could still be identified and treated as if

partitioned by examining the code. With automation, CAPTools only stores a list of partitioned variables for the current partition, meaning that implicitly partitioned variables will have to be identified by testing the execution control masks associated with the communicated data. Note that information relating to previous partitions cannot be used to convert communications of implicitly partitioned data into DLB communications, as the implicit partition information is not stored. A call to the procedure FINDIMPLICPART is used to perform an interprocedural search using True and Routine Input dependencies (Sections B.6.1 and B.6.7) for all assignments of communicated data, returning the relevant details that are used to set up the DLB parameters, as illustrated in Figure 5.14.

```
(* Convert communication in normal manner unless indicated otherwise )
(* IMPLICIT_PARTLIST=TRUE indicates conversion with 'offsets' from implicit partition *)
(* SPECIALCOMM=TRUE indicates conversion into 'special' communication using partitioned *)
(* mask *)
IMPLICIT_PARTLIST:=FALSE;
SPECIALCOMM:=FALSE;
IF (PARTLIST = NIL) THEN
   BEGIN
   (* Communicated data not in partition list for this routine *)
   (* Check whether communicated data is implicitly partitioned, or if it is surrounded by a *)
   (* partitioned mask *)
   IMPLICIT_PARTLIST:=TRUE;
   FINDIMPLICPART(CROUTINE,CCOMMAND,SYMBOL,
                  MASKEDCOMM,PARTLIST,MODNONLOOP,DIVNONLOOP,
                  MAXOFFSETNONLOOP,MINOFFSETNONLOOP,MODCONST,DIVCONST,
                  MAXOFFSETCONSTANT,MINOFFSETCONSTANT,INDEX,TRUE);
   IF (MASKEDCOMM <> NIL) THEN
      (* Partition not found, but communication found to be masked in current partition *)
      SPECIALCOMM:=TRUE;
   END
ELSE
   BEGIN
   (* Use the partition information for the communicated data from the partition list of this *)
   (* routine *)
   MODNONLOOP:=PARTLIST^.MODDIVOFFPTR^.MODNONLOOP;
   DIVNONLOOP:=PARTLIST^.MODDIVOFFPTR^.DIVNONLOOP;
   MODCONST:=PARTLIST^.MODDIVOFFPTR^.MODCONST;
   DIVCONST:=PARTLIST^.MODDIVOFFPTR^.DIVCONST;
   END;
```

Figure 5.14: If the communicated data is not found in routine's partition list, then an implicit partition may be found using FINDIMPLICPART, or the value of FIRST may be determined for use in 'special' DLB communications.

The FINDIMPLICPART procedure looks at all of the masked assignment statements for the communicated data (Figure 5.15). The communication requests

are migrated up the code and then merged (Section B.9.1), meaning that the data communicated in a single call may have several assigners and usages. This procedure examines the relationship between the implicitly partitioned index of each assignment statement and the expression used in the execution control mask for that assignment statement. If there is a linear relationship between an index of an assignment statement and the expression in its mask, then it is possible to identify which processor owns the assigned data. The symbolic factor and remainder of the index factorised by the mask expression must be loop invariant. If such a relationship exists with all of the assigners of the communicated data, where they are either a subset of a superset of each other, and no relationship is contradictory, then the data can be treated as if partitioned. If the relationship between the assignment index and the mask of just one of the assigner statements does not fit (i.e. is not linear, or is not a subset or superset of the other relationships), then CAPTools will have already broadcast this data since the location of it is unknown (Section B.9.1.3). Note that the locality of communications involving implicitly partitioned data tend to be relatively close to the assignment statement(s) of that data (as illustrated in Figure 4.9).

The algorithm used here follows on from the algorithms already used by CAPTools, since it has already been determined whether there was any need to communicate this data. This implies that if a communication (excluding a Broadcast) was generated then the relationship that was used can be determined again.

```
PARTINDEX:=0;
(* Find first assigner of the communicated data (including those in caller/called routines) *)
   (* For each index_J *)
      (* If linear relationship between index_J and assignment mask M *)
         PARTINDEX:=index_J;
         (* Find F (factor) and R (remainder) where F.index_J+R=M *)
         MAXOFFSET=R;
         MINOFFSET=R;
         (* For ALL assigners of communicated data (including those in caller/called routines) *)
            (* Extract index_J (or extract with MOD/DIV) *)
            (* Calculate F' and R' where F'.index_J'+R'=M' *)
            IF (F' <> F) OR (M'partition <> Mpartition) OR
                  NOT(NONLOOPCONSTANT(R' – R)) THEN
               (* Remainder is loop variant *)
               PARTINDEX:=0
            ELSE
               MAXOFFSET:=MAX(MAXOFFSET,R');
               MINOFFSET:=MIN(MINOFFSET,R');
```

Figure 5.15: Pseudo algorithm used to evaluate the communication 'offsets' that determine LOWLIM and HIGHLIM.

If the call to FINDIMPLICPART returns a pseudo partition, then this information is used to set up the DLB call as above. The communication offsets (Section 3.3.1.2) are also evaluated inside FINDIMPLICPART, which are then used to determine the values of LOWLIM and HIGHLIM, as illustrated in Figure 5.16. The expression for LOWLIM is set to be the summation of the lower processor partition range limit in the Staggered Dimension (CAP_L) and the value returned in MINOFFSET. Similarly, the expression for HIGHLIM is set to be the summation of the upper processor partition range limit in the Staggered Dimension (CAP_H) and the value returned in MAXOFFSET.

```
(* Add LOWLIM and HIGHLIM to communication tree *)
IF (IMPLICIT_PARTLIST) THEN
   BEGIN
   (* Set up LOWLIM by adding together CAP_L and the calculated LOWOFFSET *)
   NEWTREE(DLB_LOWLIM_TREE);
   NEWTREE(DLB_LOWLIM_TREE^.LEFT);
   BUILDNON(PARTLIST^.MINSYMB,PARTLIST^.MINTREE,PARTLIST^.ROUTINE^.START,
                PARTLIST^.ROUTINE,NONLOOP_LOWOFFSET);
   ADDLIST(NONLOOP_LOWOFFSET,NONLOOP_LOWOFFSET,MINOFFSETNONLOOP,
              1,1,9999,NIL,FALSE);
   BUILDTREE(NONLOOP_LOWOFFSET,MINOFFSETCONSTANT,CROUTINE,
                NIL,CCOMMAND,NIL,DLB_LOWLIM_TREE^.LEFT,FALSE,TRUE);
   DLB_LOWLIM_TREE^.RIGHT:=TREE^.RIGHT;
   TREE^.RIGHT:=DLB_LOWLIM_TREE;
   TREE:=DLB_LOWLIM_TREE;
   (* Set up HIGHLIM by adding together CAP_H and the calculated HIGHOFFSET *)
   NEWTREE(DLB_HIGHLIM_TREE);
   NEWTREE(DLB_HIGHLIM_TREE^.LEFT);
   BUILDNON(PARTLIST^.MAXSYMB,PARTLIST^.MAXTREE,PARTLIST^.ROUTINE^.START,
                PARTLIST^.ROUTINE,NONLOOP_HIGHOFFSET);
   ADDLIST(NONLOOP_HIGHOFFSET,NONLOOP_HIGHOFFSET,MAXOFFSETNONLOOP,
              1,1,9999,NIL,FALSE);
   BUILDTREE(NONLOOP_HIGHOFFSET,MAXOFFSETCONSTANT,CROUTINE,
                NIL,CCOMMAND,NIL,DLB_HIGHLIM_TREE^.LEFT,FALSE,TRUE);
   DLB_HIGHLIM_TREE^.RIGHT:=TREE^.RIGHT;
   TREE^.RIGHT:=DLB_HIGHLIM_TREE;
   TREE:=DLB_HIGHLIM_TREE;
   END;
```

Figure 5.16: Setting up the LOWLIM and HIGHLIM parameters when the communicated data is implicitly partitioned, where any offsets determined in FINDIMPLICPART are included in the expression.

If the call to FINDIMPLICPART cannot find an implicit partition, then an attempt is made to try to identify whether the assignment statement(s) of the communicated data are made on a specific processor (i.e. data only communicated by a processor owning a specific value in the Staggered Dimension, and not within a range). The processor ownership mask identifying where the assignment was made will be returned. If the execution control mask of the assignment statement is related to the current dimension (the Staggered Dimension), then the communication will be converted into a special DLB communication (Section 3.3.4). The value of FIRST is set to the constant value in the execution control mask (Figure 4.8), and STAG_STRIDE is set to 0 (Figure 5.17). Note that for simplicity, LOWLIM and HIGHLIM are set to the lower and upper processor partition range limits respectively although they are not actually used internally if STAG_STRIDE=0.

```
IF (SPECIALCOMM) THEN
   BEGIN
   (* Set up tree for FIRST *)
   BUILDTREE(MASKEDCOMM^.LINK^.NONLOOP,MASKEDCOMM^.LINK^.CONSTANT,
                  CROUTINE,NIL,CCOMMAND,NIL,DLB_FIRST_TREE^.LEFT,FALSE,TRUE);
   (* Set up tree for STAG_STRIDE *)
   NEWTREE(DLB_STAG_STRIDE_TREE^.LEFT);
   SETHAS(DLB_STAG_STRIDE_TREE^.LEFT^.SYMB,'0    ');
   END;
```

Figure 5.17: Setting up the FIRST and STAG_STRIDE parameters for a 'special' DLB communication.

5.7 Inserting The DLB Code

To fully implement the DLB Staggered Limit Strategy the code that actually performs the DLB needs to be inserted into the parallel code. Section 5.7.1 discusses the initialisation of the parallel code to execute in DLB mode so that the DLB communications will operate correctly. Section 5.7.2 discusses the automatic implementation of the basic DLB code, where details relating to the construction of migration calls, and the duplication of overlap communications, are discussed in Sections 5.8 and 5.10 respectively.

5.7.1 Initialising DLB Mode

Each processor needs to know who their potential neighbours are in each Non-Staggered Dimension, along with the staggered processor partition range limits of those neighbours. This information is internally stored within the ALLNEIGHBOURS and CAP_DLB_PROCLIMITS data structures, which are set up in the CAP_DLB_SETALLNEIGHBOURS and CAP_DLB_SETUPLIMITS utilities respectively (see Section 3.2.1 and 3.2.2). The calls to these utilities therefore have to be inserted into the parallel application code.

CAPTools currently sets up the initialisation of the non-DLB parallel code by placing a call to CAP_INIT at the end of the declaration statements in the Main program. Similarly, a call to set up and initialise the DLB parallel code must be

made as early on in the code as possible. This can therefore be done by placing a call to CAP_DLB_SETUPALLNEIGHBOURS after the call to CAP_INIT which is currently at the end of the declaration list (Figure 5.18), before any executable statements that involve partitioned data. Figure 5.19 illustrates how this new command can be inserted into the code using CAPTools, where a new command (NEWCOM) is inserted at the end of the declaration list in the Main program using ADDDECLCOMMAND (Table B.3). Note that ADDDECLCOMMAND can be used to declare (or initialise) any new DLB variables into a specified routine.

Additionally, in order to correctly set up the DLB parallel code, information relating to the processor partition range limits of neighbouring processors must be made available for use internally. The processor partition range limits are set up in CAPTools using either a call to CAP_SETUPPART or CAP_SETUPDPART. Therefore the information needed to operate in DLB mode can only be set up once the partitions have been constructed, which means identifying the setup call and constructing an additional call to CAP_DLB_SETUPLIMITS immediately after. Figure 5.20 illustrates how to construct this call, where every command is examined since the processor partition range limits may be set up anywhere within the code and not simply in the Main program.

```
Program Main
declaration 1
....
declaration n
call cap_init
call cap_dlb_setupallneighbours
....
call cap_setupdpart(1,NJ,cap2_low,cap2_high,2)
call cap_dlb_setuplimits(cap2_low,cap2_high,2)
call cap_setupdpart(1,NI,cap1_low,cap1_high,1)
call cap_dlb_setuplimits(cap1_low,cap1_high,1)
....
```

Figure 5.18: Example setting up the parallel code to execute in DLB mode.

```
(* Initialise DLB mode at the end of the Main programs' declaration statements *)
ADDDECLCOMMAND(MAINPROG,NEWCOM);
SETHAS(NEWCOM^.LINK^.SYMB,'CALL');
NEWTREE(NEWCOM^.LINK^.LEFT);
SETHAS(NEWCOM^.LINK^.LEFT^.SYMB,'CAP_DLB_SETUPALLNEIGHBOURS');
```

Figure 5.19: Inserting a new command at the end of the declaration list for a specified routine.

```
(* Examine every command in code and find command that sets up the processor *)
(* partition range limits *)
IF (CCOMMAND^.LINK^.SYMB^.KIND = KEYCALL) AND
     (COMPSTRING(CCOMMAND^.LINK^.LEFT^.SYMB^.NAME,
       'CAP_SETUPDPART',14)) THEN
   BEGIN
   (* Several partitions have been created using *)
   (* CALL CAP_SETUPDPART(loassn,hiassn,lopart,hipart,iaxes) *)
   (* Extract lopart – lower processor partition range limit *)
   LOW:=CCOMMAND^.LINK^.LEFT^.RIGHT^.RIGHT^.RIGHT^.LEFT;
   (* Extract hipart – upper processor partition range limit *)
   HIGH:=CCOMMAND^.LINK^.LEFT^.RIGHT^.RIGHT^.RIGHT^.RIGHT^.LEFT;
   (* Extract iaxes – the partition number *)
   PART_NUM:=CCOMMAND^.LINK^.LEFT^.RIGHT^.RIGHT^.RIGHT^.RIGHT^.RIGHT^.LEFT;
   (* Set up the tree for the new command immediately after the found command *)
   ...
   END
ELSE IF(CCOMMAND^.LINK^.SYMB^.KIND = KEYCALL) AND
   (COMPSTRING(CCOMMAND^.LINK^.LEFT^.SYMB^.NAME,'CAP_SETUPPART',13)) THEN
   BEGIN
   (* Only one partition has been created using *)
   (* CALL CAP_SETUPPART(loassn,hiassn,lopart,hipart) *)
   (* Extract lopart – lower processor partition range limit *)
   LOW:=CCOMMAND^.LINK^.LEFT^.RIGHT^.RIGHT^.RIGHT^.LEFT;
   (* Extract hipart – upper processor partition range limit *)
   HIGH:=CCOMMAND^.LINK^.LEFT^.RIGHT^.RIGHT^.RIGHT^.RIGHT^.LEFT;
   (* Set the partition number to 1, since 1D partition used *)
   INT2TOKEN(1,PART_NUM_TOKEN);
   (* Set up the tree for the new command immediately after the found command *)
   ...
   END
```

Figure 5.20: Identifying calls that determine the processor partition range limits, which are used to construct the parameters needed for the call to CAP_DLB_SETUPLIMITS.

5.7.2 The Underlying DLB Implementation Code

The parallel code is now capable of operating when non-coincidental limits are used, since each processor knows the limits of each of its potential neighbours and

can communicate using the inserted DLB communications. The remaining Sections of this Chapter will now concentrate on the automation process involved with load redistribution, how to calculate the new processor partition range limits and migrating the load to ensure processor ownership.

Given the selected DLB Routine and Loop from the DLB Browser window (Section 5.2), it is possible to place the underlying DLB code at the start of the DLB Loop (Figure 5.21). This consists of a call to CAP_DLB_DECIDE which stops timing the load imbalance in the current iteration and determines whether or not to redistribute the workload (Table 3.13). A call to CAP_DLB_START_REBAL is then needed to find the new partition and decide whether the new partition should be implemented, after which the calls to initiate migration in each partitioned dimension are required. Additionally, a call to the CAP_DLB_START_TIMER utility needs to be inserted in order to start timing the contents of the load imbalanced loop after the load has been redistributed.

```
CALL CAP_DLB_DECIDE(...)
IF (CAP_DLB_PERFORM_REBAL) THEN
   CALL CAP_DLB_START_REBAL(...)
   IF (CAP_DLB_MIGRATE_DIM(1)) THEN
      migration calls in dimension 1
   END IF
   IF (CAP_DLB_MIGRATE_DIM(2)) THEN
      migration calls in dimension 2
   END IF
   IF (CAP_DLB_MIGRATE_DIM(3)) THEN
      migration calls in dimension 3
   END IF
   IF (CAP_DLB_MIGRATE_DIM(1) .OR.
         CAP_DLB_MIGRATE_DIM(2) .OR.
          CAP_DLB_MIGRATE_DIM(3)) THEN
      CALL NEW2OLD_LIMITS()
      duplicated overlap communications
      CALL CAP_DLB_STOP_REBAL(...)
   ELSE
       CAP_DLB_REBAL_TIME=CAP_DLB_PREV_REBAL_TIME
   END IF
END IF
CALL CAP_DLB_START_TIMER(...)
```

Figure 5.21: The underlying DLB implementation code that is placed at the beginning of an iteration of the DLB Loop.

Depending on the selected DLB Loop and its loop nesting, the fragment of underlying DLB implementation code can be inserted into either an identified

block or a newly created block. The code used to insert this fragment of code is shown in Figure 5.22.

To illustrate the need to test the loop nesting, consider the situation in which either the DO 10, or the 200 loop is selected as the DLB Loop (Figure 5.23). Although either the DO 10 or the 200 loop can be selected using the DLB Browser window (Figure 5.3), the selected loop head is the same within CAPTools (i.e. 200 DO 10 I=1,N) and so CAPTools needs to be able to distinguish between the two. The nesting (CNEST) for this loop head will always be stored as 200-GOTO -> DO 10 I=1,N -> Nil within CAPTools, with the innermost (last) entry always being the DO Loop (Section B.5). This means that if the DLB Loop is the DO Loop then a new block will be generated within this loop immediately after the DLB Loop head, as CNEST^.NEXT will be NIL. If the selected DLB Loop is the 200-GOTO loop then the nesting list will not have been fully traversed and a new block will be generated before the DO Loop. The label from the 200-GOTO loop will automatically be transferred onto this new block inside the call to CREATEBLOCK (Section B.6.10.3).

```
(* Set up the current block *)
CBLOCK:=DLBLOOP^.HEAD;
IF (CBLOCK^.COMMAND^.LINK^.SYMB^.KIND IN [KEYDO,KEYDOWHILE]) THEN
   BEGIN
   CNEST:=CBLOCK^.NESTING;
   WHILE (CNEST^.LOOPINFO <> DLBLOOP) DO
      CNEST:=CNEST^.NEXT;
   (* Create a new block (NBLOCK) in which to place the fragment of DLB code *)
   IF (CNEST^.NEXT <> NIL) THEN
      (* This is not the DO Loop – generate new block before the current block *)
      CREATEBLOCK(DLBROUTINE,CBLOCK,TRUE,TRUE,NBLOCK)
   ELSEIF (CNEST^.NEXT = NIL) THEN
      (* This is the DO Loop – generate new block within the current block *)
      CREATEBLOCK(DLBROUTINE,CBLOCK,FALSE,FALSE,NBLOCK);
   END
ELSE
   (* Put the fragment of DLB code into the current block *)
   NBLOCK:=CBLOCK;
(* Wire up code so that the fragment of DLB code is at the top of the block *)
```

Figure 5.22: The code used to determine the block containing the fragment of underlying DLB implementation code.

Original parallel code example:

```
        NITER=1
200     DO 10 I=...
           ....
           CALL SOLVER1()
           ....
10      CONTINUE

        CALL SOLVER2()
        IF (NITER.LE.MAX_ITER) THEN
           A=...
           B=
           GOTO 200
        END IF
```

The 200-Goto loop is the DLB Loop:

```
        NITER=1
200     Fragment of DLB code
        DO 10 I=...
           ....
           CALL SOLVER1()
           ....
10      CONTINUE

        CALL SOLVER2()
        IF (NITER.LE.MAX_ITER) THEN
           A=...
           B=
           GOTO 200
        END IF
```

The Do 10 loop is the DLB Loop:

```
        NITER=1
200     DO 10 I=...
           Fragment of DLB code
           ....
           CALL SOLVER1()
           ....
10      CONTINUE

        CALL SOLVER2()
        IF (NITER.LE.MAX_ITER) THEN
           A=...
           B=
           GOTO 200
        END IF
```

Figure 5.23: Example illustrating the need to consider the loop nesting when deciding where to place the code shown in Figure 5.21.

Any DLB variables that are introduced into the parallel application need to be declared in the DLB Routine, and some of them also need to be initialised. As illustrated in Figure 5.19, any new declarations can easily be added to a specified

routine (the DLB Routine in this case), where any initialisations can be made at the end of that declaration list.

Note that if the load were redistributed at the end of the DLB Loop then more testing would be required in terms of this entire algorithm. Communications in the DLB Loop itself would also need to be considered for duplication (Section 5.10), and the test that compares the call paths of the usage statement and the redistribution statement would involve additional work. The example shown in Figure 5.24 illustrates the fact that when redistributing at the end of the DLB Loop (at REDISTR. B), the communication on statement S4 will need to be duplicated, where this is not the case if redistributing at the beginning of the DLB Loop (at REDISTR. A). In the former instance, if this communication is not duplicated then the usage of T in statement S12 will be using the overlap data that was updated using a previous distribution.

		If redistributed at	
		REDISTR. A	REDISTR. B
S1	do		
S2	do j=		
	REDISTR. A	*update distribution*	*- na -*
S3	T(..,j)=	use new partition	use old partition
S4	**comm(T)**	use new partition	use old partition
S5	...=T(..,j)	use new partition	use old partition
S6	**...=T(..,j-1)**	use new partition	use old partition
	REDISTR. B	*- na -*	*update distribution*
S7	end do		
S8	end do		
S9	do		
S10	do j=		
S11	...=T(..,j)	use new partition	use new partition
S12	**...=T(..,j-1)**	use new partition	***use old partition***
S13	end do		
S14	end do		

Figure 5.24: Example illustrating the need to duplicate the communication in S4 when the workload is redistributed at REDISTR. B (at the end of the DLB Loop) due to the usage of the variable T in statement S12.

5.8 Inserting The Migration Calls

This Section concentrates on the automation process involved in identifying data that must be migrated and how the necessary migration calls are constructed. As discussed in Section 2.8, data needs to be migrated from one processor to another in order to implement the newly calculated distribution, which is achieved using the migration calls discussed in Section 3.7. Every item of data that is affected by the new partition needs to be migrated.

5.8.1 Identify Data To Be Migrated

When an array is partitioned, this implies that the range of data between the processor partition range limits is owned on a processor, therefore when the limits are changed then the data affected by the new distribution will need to be migrated. An array is affected by the new distribution if it is assigned before redistribution and used after redistribution, since the array will be assigned on one processor before redistribution and may then be used on a different processor after redistribution. This is also true for data that is assigned in one iteration of the DLB Loop and used in a subsequent iteration.

Every partitioned variable in a routine will be stored in the routine's partition list (ROUTINE^.PARTITION), where it is known that this list includes partitioning information relating to local variables in called routines (partitions that have been inherited by the calling routine, see Section B.7.2), and so it is possible to use this list as a basis for automatically constructing the migration calls. It is possible to detect all of the partitioned variables that need to be migrated using the partition list for the Main program, as this list will include those variables that are partitioned in every other routine including the DLB Routine. This will ensure, for instance, that even those partitioned variables in Figure 5.25 that are locally declared in Sub1 and only used in Sub3 will be considered for migration. If the DLB Routine partition list were used to detect variables to migrate then this case would not be considered, since the partitioned variables in Sub3 would not be included in the DLB partition list (as Sub3 is not

called from there). Using the partition list only guarantees the identification of partitioned variables to migrate, it does not identify unpartitioned variables to migrate, this is dealt with in Section 5.8.4. Note that most statements in each routine of this example are expressed in terms of the statement number followed by the routine name (i.e. S2_Sub1 and SN_Sub1 are the second and N^{th} statements in subroutine Sub1).

```
Program Main
S1_Main
...
call Sub1
call Sub2
call Sub3

SN_Main
End
```

```
Subroutine Sub1
S1_Sub1
S2_Sub1
...
u(...)=
comm(u(..),..)

SN_Sub1
End
```

```
Subroutine Sub2
S1_Sub2
S2_Sub2
...
call Sub5
call SubDLB
...
SN_Sub2
End
```

```
Subroutine Sub3
S1_Sub3
S2_Sub3
...
...=u(...)


SN_Sub3
End
```

```
Subroutine SubDLB
S1_SubDLB
call Sub4
...

DLB Loop
C Redistribution

...
call Sub5
SN_SubDLB
End
```

```
Subroutine Sub4
S1_Sub4
S2_Sub4
...
SN_Sub4
End
```

```
Subroutine Sub5
S1_Sub5
S2_Sub5
...
SN_Sub5
End
```

Figure 5.25: Example illustrating a code in which the redistribution occurs in the SubDLB, which is called from Sub2 that is called from the Main program.

Since arrays can be partitioned in any number of dimensions, each partitioned dimension is processed separately, with the Staggered Dimension being processed last. The dimension being processed is known as the Migration Dimension, for which all of the necessary migration calls are generated followed by a call to CAP_DLB_REASSIGNLOWHIGH, which updates the processor partition range limits for the Migration Dimension. This ensures that the data is moved and the limits updated for each Migration Dimension before processing subsequent dimensions.

As well as using the current partition details in the Staggered Dimension (SDPART) for the Main routine, the generation of migration calls requires information relating to previous partitions generated in a Non-Staggered Dimension (NSPART), information that CAPTools does not store. The previous partition details of a routine can be extracted from a new data structure called OLDPARTITIONLIST (Section 5.4). The GEN_NS_MIG_CALLS procedure and the GEN_SD_MIG_CALLS procedure are used to generate all of the migration calls for the Non-Staggered Dimensions and for the Staggered Dimension respectively as seen in Figure 5.26. Both the MAINROUTINE and the DLBROUTINE will need to be passed into these procedures since the partition information (and any declaration information) is obtained from the Main program, and any generated statements need to be inserted into the DLB Routine. The partition number of the dimension being processed (PARTITION_NUMBER) is also passed in, where it is used to construct the Migration Dimension number (MD) in the generated migration call (Section 3.7). Note that the order in which data is migrated is arbitrary, implying that it makes no difference what order the Non-Staggered Dimensions are processed in (Section 4.7).

```
(* NUM_OF_NS_PARTITIONS equals the number of Non-Staggered partitions *)
(* Previous partitions stored in reverse order (i.e. 1st partition is last in list) *)
NSCOUNT:=0
(* Process the partition lists of each of the Non-Staggered Dimensions *)
COLDPARTITIONLIST:=MAINROUTINE^.OLDPARTITIONLIST;
WHILE (COLDPARTITIONLIST <> NIL) DO
   BEGIN
   (* Process the previous partition list of the Main program for this Migration Dimension *)
   NSPART:=COLDPARTITIONLIST^.PARTITION;
   (* Process the partition list of this Non-Staggered Dimension *)
   PARTITION_NUMBER:=NUM_OF_NS_PARTITIONS-NSCOUNT;
   (* Generate all migration calls in the DLB Routine based on the partition list of the *)
   (* Main program *)
   GEN_NS_MIG_CALLS(MAINROUTINE,DLBROUTINE,NBLOCK,NEWCOMMAND,
                              NSPART,PARTITION_NUMBER,NUM_OF_NS_PARTITIONS);
   NSCOUNT:=NSCOUNT+1;
   (* Process the partition list of the next Non-Staggered Dimension, if there is one *)
   COLDPARTITIONLIST:=COLDPARTITIONLIST^.NEXT;
   END;
(* Process the current partition list of the Main program for the Staggered Dimension *)
SDPART:=MAINROUTINE^.PARTITON;
PARTITION_NUMBER:=NSCOUNT+1;
(* Generate all migration calls for the Staggered Dimension *)
GEN_SD_MIG_CALLS(MAINROUTINE,DLBROUTINE,NBLOCK,NEWCOMMAND,
                           SDPART,PARTITION_NUMBER)
```

Figure 5.26: Code used to process the Non-Staggered Migration Dimensions followed by the Staggered Dimension, where all of the migration calls are generated for the processed dimension along with the call to update that dimensions processor partition range limits.

5.8.2 Constructing The Migration Calls

There are two distinct migration calls, one for migrating to an immediate neighbour in which the internal communications of the migration call are not affected by the staggered limits, and the other is used when migrating over the non-coincidental limits (Section 3.7). If migrating in a Non-Staggered Dimension where the data is not also partitioned in the Staggered Dimension, or if migrating in the Staggered Dimension, then a call to CAP_MIGRATE() is used, otherwise a call to CAP_DLB_MIGRATE() is used, as illustrated in Figure 5.27.

```
if migrating in the Staggered Dimension then use

   CAP_MIGRATE(A,START_IND,STRIDE,S1,NS1,S2,NS2,S3,NS3,
                                  S4,NS4,S5,NS5,S6,NS6,ITYPE,MD)

else if migrating in a Non-Staggered Dimension then
   if data also partitioned in the Staggered Dimension then use

      CAP_DLB_MIGRATE(A,START_IND,STRIDE,STAG_IND,
                                  STAG_STRIDE,S1,NS1,S2,NS2,S3,NS3,
                                  S4,NS4,S5,NS5,S6,NS6,ITYPE,MD)

   else use

      CAP_MIGRATE(A,START_IND,STRIDE,S1,NS1,S2,NS2,S3,NS3,
                                  S4,NS4,S5,NS5,S6,NS6,ITYPE,MD)

   end if
end if
```

Figure 5.27: Pseudo code used to determine the new call name for a converted communication.

An example illustrating the migration call name for the variable T when processing each partitioned dimension is shown in Figure 5.28, where the sixth dimension has been partitioned first, the second dimension has been partitioned second and the third dimension has been partitioned last (i.e. the Staggered Dimension).

REAL T(l_1:h_1, l_2:h_2, l_3:h_3, l_4:h_4, l_5:h_5, l_6:h_6, l_7:h_7)

P2 P3 (SD) P1

Pass	Index	Lower Limit	Upper Limit	DIV	MOD
1	6	CAP1_LOW	CAP1_HIGH	S_6	S_7
2	2	CAP2_LOW	CAP2_HIGH	S_2	S_3
3	3	CAP3_LOW	CAP3_HIGH	S_3	S_4

Where the stride of index i can be expressed as:

$$S_i = \prod_{j=1}^{i-1} \left(h_j - l_j + 1 \right)$$

```
Processing Migration Dimension=1
    CALL CAP_DLB_MIGRATE(...)

Processing Migration Dimension=2
    CALL CAP_DLB_MIGRATE(...)

Processing Migration Dimension=3
    CALL CAP_MIGRATE(...)
```

Figure 5.28: Example illustrating the migration call name for the variable T that has been partitioned as shown.

If the data being migrated is not already declared in the DLB Routine then it is added to the DLB Routine declaration list using the INLINECOMMONS procedure (Section B.6.10.3). Note that at present the common blocks and local variables of called routines are internally stored within a given routine and are not generated in the final code, which means that any required implicit common blocks of the Main program can be copied into the DLB Routine, allowing it to access all the necessary variables.

5.8.2.1 Setting Up The Starting Address For The Migrated Data

Having identified what data needs to be migrated (Section 5.8.1), the next stage is to extract or calculate the components needed to generate the migration call

parameters. The migrated data needs to be communicated from a certain location in memory, and so the starting address (A in Figure 5.27) of this initial location is passed into the migration utility. As stated in Section 3.7.1, the starting location of each index of the migrated data will be either the low declared limit of the index or the lower processor partition range limit. If the index is not partitioned, or is partitioned either in the current Migration Dimension or in the Staggered Dimension, then the low declared limit is used, otherwise the lower processor partition range limit is used (Figure 5.29). The low declared limit can be extracted from the declaration statement and the lower processor partition range limit can be extracted from the partition information for the particular index.

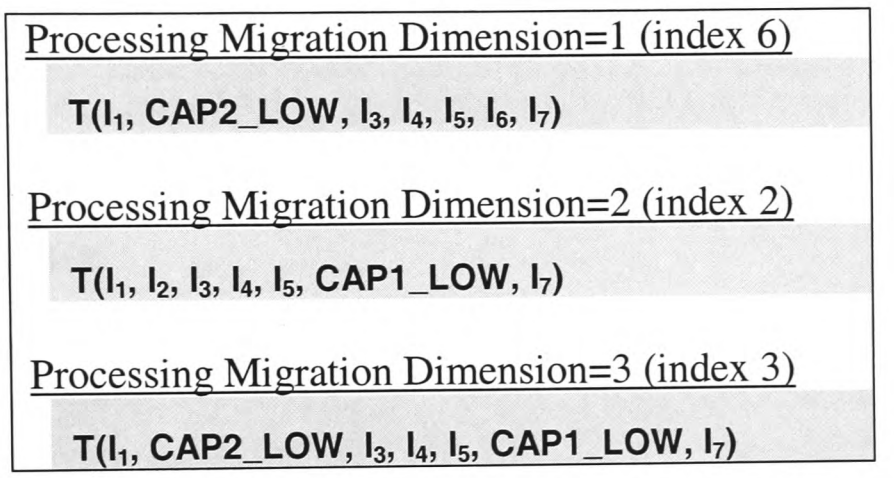


Figure 5.29: Example illustrating the starting address for the migrated variable T.

When processing a particular Migration Dimension, two flags can be set up so that Staggered Dimension and the Migration Dimension can be easily identified (IS_STAG and IS_MIG respectively). These two flags can also be used to indirectly identify any Non-Staggered Dimensions other than the Migration Dimension, making it possible to distinguish between the different types of partitions so that a particular partition can be singled out. NSPART(1) and NSPART(2) contain the partition information relating to the first and second Non-Staggered Dimensions respectively and SDPART contains the current partition information. Given the value of IS_STAG and IS_MIG for each particular Migration Dimension for the example in Figure 5.29, the lower processor partition range limit would be used for both of the Non-Staggered Dimensions when processing the Staggered Dimension (Table 5.1). The pseudo code used to determine the starting address of the migrated data is shown in Figure 5.30.

Processing Migration Dimension=1 (index 6)

Partition:	IS_STAG:	IS_MIG:
NSPART(1)	False	True
NSPART(2)	False	False
SDPART	True	False

Processing Migration Dimension=2 (index 2)

Partition:	IS_STAG:	IS_MIG:
NSPART(1)	False	False
NSPART(2)	False	True
SDPART	True	False

Processing Migration Dimension=3 (index 3)

Partition:	IS_STAG:	IS_MIG:
NSPART(1)	False	False
NSPART(2)	False	False
SDPART	True	True

Table 5.1: The values of IS_STAG and IS_MIG are given for each Migration Dimension for the migrated variable T.

```
For each index:
   If ((index is partitioned) and (not IS_MIG) and (not IS_STAG)) then
      (* Index is partitioned in a Non-Staggered Dimension other than *)
      (* the Migration Dimension *)
      Use lower processor partition range limit
   Else
      (* Index is not partitioned, or it is partitioned in the
           Migration Dimension or Staggered Dimension *)
      Use low declared limit
   End If
```

Figure 5.30: The pseudo algorithm used to determine the starting address for the migrated variable T.

5.8.2.2 Setting Up The Starting Index And Stride For The Migrated Data In The Migration Dimension And The Staggered Dimension

The migration parameters (START_IND, STRIDE, STAG_IND and STAG_STRIDE in Figure 5.27) need to be included in the migration call to enable the new starting address and amount of data to be buffered to be calculated

for each internal communication call. For both the Migration Dimension and the Staggered Dimension the starting address is the low declared limit and the stride is simply the DIV component in the MODDIVOFFPTR field of the partition in question. These parameters are stored within the PARTITION information, and so they can be extracted from the relevant partition (Figure 5.31). Note that the starting address and stride for the Staggered Dimension is used only when constructing CAP_DLB_MIGRATE.

Processing Migration Dimension=1 (index 6)
START_IND=I_6
STRIDE=S_6
STAG_IND=I_3
STAG_STRIDE=S_3

Processing Migration Dimension=2 (index 2)
START_IND=I_2
STRIDE=S_2
STAG_IND=I_3
STAG_STRIDE=S_3

Processing Migration Dimension=3 (index 3)
START_IND=I_3
STRIDE=S_3

```
For each index:
   If (index is partitioned) then
      If (IS_MIG) then
         (* Use the partition details of the Migration Dimension *)
         Set START_IND to the low declared limit of this index
         Set STRIDE to the DIV component of this partition
      End If
      If (IS_STAG and not(IS_MIG)) then
         (* Use the partition details of the Staggered Dimension *)
         Set STAG_IND to the low declared limit of this index
         Set STAG_STRIDE to the DIV component of this partition
      End If
   End If
```

Figure 5.31: Example illustrating the values of START_IND, STRIDE, STAG_IND and STAG_STRIDE for the migrated variable T, along with the pseudo algorithm used to determine these parameters.

5.8.2.3 Setting Up The Stride And Number Of Strides

The index of the Migration Dimension and the index of the Staggered Dimension are represented in the migration call by the parameters discussed in Section 5.8.2.2. The remaining indices must also be represented in the migration call if data is to be migrated (internally buffered and then communicated), hence the need to pass in the stride and number of strides for each remaining index (S and NS in Figure 5.27).

The remaining indices will either be partitioned in a Non-Staggered Dimension (other than the Migration Dimension itself), or they will contain a contiguous section of memory in which each different index being processed can be identified by its stride, where the number of strides between contiguous sections of data indicates the amount of data to be migrated for that index. If an index is not partitioned then all of the data in that index will need to be migrated, whereas if the index is partitioned then only the data owned by the processor will be migrated, meaning that NS for such an index will be its processor partition range. Using T in Figure 5.28 as an example, when migrating in dimension 1 all of the data in index 1 would need to be migrated where its stride (S1) is 1 and $NS1=h_1-l_1+1$ since this is a contiguous section in memory. When dealing with the second index which is in a Non-Staggered Dimension the stride of index 2 would need to be extracted from its partition details ($S2=S_2$) along with the number of strides NS2=CAP2_HIGH-CAP2_LOW+1.

The remaining indices to be included in the migration call can be grouped together (if not partitioned) to form a paired-index, such that all of the indices between partitions can be treated as a single index. Processing each paired-index allows more indices to be buffered inside the migration call, making it less likely to have to place the migration call within buffering loops. As with a single index, each paired-index must have its own stride which is equivalent to the stride of the first index in the group of indices being processed. The number of strides for the paired-index can then be set to the entire contiguous length of those indices, which is the product of their dimensions. For example, index 4 and 5 are adjacent indices of T that can be paired together since neither is partitioned, where S of the

paired-index is that of index 4 (S_4) and NS is set to $(h_4-l_4+1)*(h_5-l_5+1)$, as illustrated in Figure 5.32.

Processing Migration Dimension=1 (index 6)

S1=S_1, NS1=(h_1-l_1+1),
S2=S_2, NS2=(CAP2_HIGH-CAP2_LOW+1),
S3=S_4, NS3=$(h_4-l_4+1)*(h_5-l_5+1)$,
S4=S_7, NS4=(h_7-l_7+1),
S5=1, NS5=1,
S6=1, NS6=1

Processing Migration Dimension=2 (index 2)

S1=S_1, NS1=(h_1-l_1+1),
S2=S_4, NS2=$(h_4-l_4+1)*(h_5-l_5+1)$,
S3=S_6, NS3=(CAP1_HIGH-CAP1_LOW+1),
S4=S_7, NS4=(h_7-l_7+1),
S5=1, NS5=1,
S6=1, NS6=1

Processing Migration Dimension=3 (index 3)

S1=S_1, NS1=(h_1-l_1+1),
S2=S_2, NS2=(CAP2_HIGH-CAP2_LOW+1),
S3=S_4, NS3=$(h_4-l_4+1)*(h_5-l_5+1)$,
S4=S_6, NS4=(CAP1_HIGH-CAP1_LOW+1),
S5=S_7, NS5=(h_7-l_7+1),
S6=1, NS6=1

Figure 5.32: Example illustrating the values of S and NS for the migrated variable T.

Figure 5.33 shows the pseudo algorithm used to determine the values of S_{PI} and NS_{PI} for each paired-index (PI) of the migration call. Each INDEX of the migrated variable is processed separately, where the stride (DIV component) and the processor partition range is used if partitioned in a Non-Staggered Dimension other than the Migration Dimension. If a processed index is partitioned then the MOD component of that partition is stored in PI_STRIDE, which is actually the stride of the next unpartitioned index (or group of indices) to be processed. The last partitioned index to be processed is also stored (LAST_PART_IND), where this is used to determine the contiguous lengths of adjacent indices which are unpartitioned. For instance, if the next unpartitioned index to be processed is not processed immediately after a partitioned index, then this implies that it can be paired with the previous index. Note that after processing every paired-index, any S and NS parameters are set to 1 by default.

Consider for example the case when generating each S and NS for the migration call in the Staggered Dimension (index 3) in Figure 5.32. The first

processed index is not partitioned and since no other index has been processed then S will be set to PI_STRIDE which equals 1, and NS will be set to (h_1-l_1+1). The second index to be processed is partitioned, but not in the Migration Dimension or the Staggered Dimension, so the index counter is increased and the second S is set to the DIV component of this index (S_2) and NS is set to (CAP2_HIGH-CAP2_LOW+1). The stride for the next unpartitioned index (PI_STRIDE) is pre-emptively set to the MOD component of this partitioned index. The third processed index is in the Migration Dimension and the Staggered Dimension, therefore PI_STRIDE is reset to the MOD component of the Staggered Dimension. The fourth processed index is not partitioned and since the previous index was partitioned, S is set to PI_STRIDE and NS is set to(h_4-l_4+1). The fifth processed index is also unpartitioned therefore this index is paired with the previous index by adjusting NS to $(h_4-l_4+1)*(h_5-l_5+1)$. The sixth index is processed in a similar manner to the second index which was also partitioned, with S set to the DIV component of the partitioned index (S_6) and NS set to (CAP1_HIGH-CAP1_LOW+1). The final index is then processed with S set to the MOD component of the partitioned index that was previously processed, and NS set to (h_7-l_7+1), after which all remaining S and NS parameters are set to 1.

```
(* Initialise the number of paired-index that have been processed *)
PI=0
(* Initialise the stride for the paired-index *)
PI_STRIDE=1
(* Initialise the last partitioned index that was processed *)
LAST_PART_IND=0
(* Process each index of the variable being migrated *)
For each INDEX of variable to be migrated:
   If (INDEX is partitioned) then
      If ((not IS_MIG) and (not IS_STAG)) then
         (* Index is partitioned in a Non-Staggered Dimension other than *)
         (* the Migration Dimension *)
         (* Process this index *)
         PI=PI+1
         Set S_PI to the DIV component of the partitioned index
         Set NS_PI to (CAP_HIGH-CAP_LOW+1) for this partition
      End If
      (* Store the last partitioned index that was processed (i.e. this index) *)
      LAST_PART_IND=INDEX
      (* The stride for the next unpartitioned index can be extracted from this *)
      (* partition *)
      PI_STRIDE=the MOD component of the partitioned index
   Else
      (* Index is not partitioned *)
      If (LAST_PART_IND+1 = INDEX) then
         (* Process this index, where last processed index was partitioned *)
         PI=PI+1
         Set S_PI to PI_STRIDE
         Set NS_PI to (h_INDEX-l_INDEX+1)
      Else
         (* Process this as a paired-index – adjust NS_PI accordingly *)
         (* Add this unpartitioned index to the current group by multiplying *)
         (* by the declared range *)
         Adjust NS_PI to be NS_PI*(h_INDEX-l_INDEX+1)
      End If
   End If
(* Set remaining S and NS parameters to 1 *)
For J=PI+1,6
   Set S_J to 1
   Set NS_J to 1
```

Figure 5.33: The pseudo algorithm used to determine the values of S and NS.

5.8.2.4 Completing The Migration Call By Setting Up The Type Of Data Being Migrated And The Migration Dimension

These last two parameters are used in the migration call to uniquely identify the type of data being migrated and the direction that the data is migrated in (ITYPE

and MD in Figure 5.27) respectively, as illustrated in Figure 5.34. The data type can easily be extracted from the declaration statement, and the Migration Dimension is passed into this procedure as PARTITION_NUMBER (Section 5.8.1). This then gives the final migration call for the variable T as that shown in Figure 5.35.

Processing Migration Dimension=1 (index 6)

ITYPE=2, MD=1

Processing Migration Dimension=2 (index 2)

ITYPE=2, MD=2

Processing Migration Dimension=3 (index 3)

ITYPE=2, MD=3

Figure 5.34: Example illustrating the values of ITYPE and MD for the migrated variable T.

Processing Migration Dimension=1 (index 6)

CALL CAP_DLB_MIGRATE(T(I_1, CAP2_LOW, I_3, I_4, I_5, I_6, I_7), I_6, S_6, I_3, S_3, S_1, (h_1-I_1+1), S_2, (CAP2_HIGH-CAP2_LOW+1), S_4, (h_4-I_4+1)*(h_5-I_5+1), S_7, (h_7-I_7+1), 1, 1, 1, 1, 2, 1)

Processing Migration Dimension=2 (index 2)

CALL CAP_DLB_MIGRATE(T(I_1, I_2, I_3, I_4, I_5, CAP1_LOW, I_7), I_2, S_2, I_3, S_3, S_1, (h_1-I_1+1), S_4, (h_4-I_4+1)*(h_5-I_5+1), S_6, (CAP1_HIGH-CAP1_LOW+1), S_7, (h_7-I_7+1), 1, 1, 1, 1, 2, 2)

Processing Migration Dimension=3 (index 3)

CALL CAP_MIGRATE(T(I_1, CAP2_LOW, I_3, I_4, I_5, CAP1_LOW, I_7), I_3, S_3, S_1, (h_1-I_1+1), S_2, (CAP2_HIGH-CAP2_LOW+1), S_4, (h_4-I_4+1)*(h_5-I_5+1), S_6, (CAP1_HIGH-CAP1_LOW+1), S_7, (h_7-I_7+1), 1, 1, 2, 3)

Figure 5.35: Final generated migration calls for the variable T.

5.8.3 Constructing The Migration Call When The Data To Be Migrated Is 1D-Mapped

If the data to be migrated is 1D-mapped inside the Main program (from which the partition details are extracted) then this means that the migration call will need to reflect this. The partition is no longer expressed in terms of a partitioned index but

in terms of a partitioned component, for which the MOD and DIV expressions can be used. The algorithm used to construct the starting address and the algorithm used to set up S and NS are amended, since these currently rely upon using a partitioned index.

In terms of setting up the starting address for the migration call (Section 5.8.2.1), the address is going to be relative to 1 which is the starting address of the 1D variable. The lower processor partition range limit will be used for those partitioned components created in a Non-Staggered Dimension other than the Migration Dimension, meaning that the starting address will need to be offset by this component (Figure 5.36). The example shown in Figure 5.36 corresponds to the example shown in Figure 5.28, where the third dimension is considered to be the Staggered Dimension. Note that the processor partition range variables for 1D-mapped references are relative to 1 and not the lower declared limit as in Figure 5.29 (i.e. CAP1_LOW in Figure 5.36 is equivalent to CAP1_LOW-l_6+1 in Figure 5.29). Additionally, the values of START_IND, STRIDE, STAG_START and STAG_STRIDE can be evaluated as shown in Figure 5.31 based on the DIV component of the partition under consideration. The algorithm used to construct the starting address is shown in Figure 5.37 (compare with Figure 5.30) and the main difference is that a specific expression is added into the starting address for a given partitioned component. Every other component in the variable is set to its low declared limit, which corresponds with the starting address of 1. Therefore no other components need to be added to the starting address (compare with the example in Figure 5.29).

$dim_1=(h_1-l_1+1)$; $dim_2=(h_2-l_2+1)$; $dim_3=(h_3-l_3+1)$;
$dim_4=(h_4-l_4+1)$; $dim_5=(h_5-l_5+1)$; $dim_6=(h_6-l_6+1)$;
$dim_7=(h_7-l_7+1)$

REAL $T(dim_1*dim_2*dim_3*dim_4*dim_5*dim_6*dim_7)$

Processing Migration Dimension=1

$T(1+(CAP2\_LOW-1)*S_2)$

Processing Migration Dimension=2

$T(1+(CAP1\_LOW-1)*S_6)$

Processing Migration Dimension=3

$T(1+(CAP2\_LOW-1)*S_2+(CAP1\_LOW-1)*S_6)$

Figure 5.36: Example illustrating the starting address for the migrated variable T that is 1D-mapped, which is identical to the starting address shown in Figure 5.29 for when T is not 1D-mapped.

```
(* Starting address is initially set to 1 *)
Examine each partition:
   If ((not IS_MIG) and (not IS_STAG)) then
      (* Index is partitioned in a Non-Staggered Dimension other than *)
      (* the Migration Dimension *)
      Add (CAP_LOW-1)*DIV component of partition into starting address term
   Else If (IS_MIG) then
      (* Set up START_IND and STRIDE *)
      If (DIV component = 1) then
         START_IND:=low declared limit
      Else
         START_IND:=1+DIV component
      End If
      STRIDE:=DIV component of partition
   Else If (IS_STAG) then
      (* Set up STAG_IND and STAG_STRIDE *)
      If (DIV component = 1) then
         STAG_IND:=low declared limit
      Else
         STAG_IND:=1+DIV component
      End If
      STAG_STRIDE:=DIV component of partition
   End If
```

Figure 5.37: The pseudo algorithm used to determine the starting address for the migrated variable T that is 1D-mapped.

The algorithm to set up S and NS (see Figure 5.33 in Section 5.8.2.3) is adjusted such that it operates in terms of the partitioned components rather than indices and this is shown in Figure 5.38. This requires identifying the unpartitioned components of an array whilst processing the partitioned

components. An unpartitioned component may exist before, in-between, or after partitioned components, as is the case for T in the example shown in Figure 5.36. The new algorithm therefore requires the partitions to be processed in ascending order of stride, where the partition with the smallest stride is processed first.

```
(* Store partitions in ascending magnitude of stride (DIV) *)
(* Initialise the number of paired-index that have been processed *)
PI=0
(* Initialise the stride for the paired-index component *)
PI_STRIDE=1
(* Process partitions in ascending magnitude of stride *)
For each PARTITION being processed:
   (* Process unpartitioned component *)
   If (DIV component <> PI_STRIDE) then
      (* There is an unpartitioned component before this partitioned component *)
      PI=PI+1
      Set S_PI to PI_STRIDE
      Set NS_PI to (DIV component/PI_STRIDE)
   End If
   (* Process this partitioned component *)
   If ((not IS_MIG) and (not IS_STAG)) then
      (* Component is partitioned in a Non-Staggered Dimension other than *)
      (* the Migration Dimension *)
      PI=PI+1
      Set S_PI to the DIV component of the partitioned variable
      Set NS_PI to (CAP_HIGH-CAP_LOW+1) for this partition
   End If
   (* The stride for the next unpartitioned index can be extracted from this partition *)
   PI_STRIDE=the MOD component of the current PARTITION being processed

(* All PARTITIONs have been processed *)
(* Process any higher component that is unpartitioned *)
If (PI_STRIDE <> 0) then
   (* There is a higher component since PI_STRIDE is not 0 *)
   PI=PI+1
   Set S_PI to PI_STRIDE
   Evaluate TOTAL_LENGTH of variable being migrated
   Set NS_PI to (TOTAL_LENGTH/PI_STRIDE)
End If

(* Set remaining S and NS parameters to 1 *)
For J=PI+1,6
   Set S_J to 1
   Set NS_J to 1
```

Figure 5.38: The pseudo algorithm used to determine the values of S and NS when the migrated data is 1D-mapped (i.e. no longer in terms of partitioned index, but partitioned component).

The stride of the first component will always be 1 (the initial value of PI_STRIDE), therefore if the stride of the first partition to be processed is 1 then

there are no unpartitioned components before it. For example, when constructing the migration call for T in Migration Dimension=1, the partition of the second Non-Staggered Dimension (whose index is 2 when not 1D mapped) is processed first, which has a stride of S_2. An unpartitioned component exists before this partition since its stride is not equal to 1, so the values of S1 and NS1 need to be set up. The value of S1 for this unpartitioned component is set to PI_STRIDE (currently set to 1), where the value of NS1 is set to the DIV component of the partition being processed divided by the value of PI_STRIDE (which in this instance equals $S_2/1=h_1-l_1+1$).

The partitioned component itself can then be processed, where the values of S2 and NS2 are set if the partition was generated in a Non-Staggered Dimension other than the Migration Dimension (i.e. both IS_MIG and IS_STAG are false). The value S2 is set to the DIV component of the partition being processed, and NS2 is set to the processor partition range. The value of PI_STRIDE is then set to the MOD of the partition being processed, as this is the stride of the next component to be processed. Continuing with the example of T, this means setting S2 to S_2 and NS2 to (CAP2_HIGH-CAP2_LOW+1), and then setting PI_STRIDE to S_3 (which is the MOD of this partition). The partition of the Staggered Dimension is the next to be processed, as its stride is less than that of the first Non-Staggered Dimension. Its stride (DIV=S_3) is the same as PI_STRIDE, meaning that there is no unpartitioned component between this and the previously processed partition. The value of PI_STRIDE is therefore set to S_4 to skip the Staggered Dimension component before processing the next partition (that of the first Non-Staggered Dimension).

The stride of the final partition (S_6) is not equal to PI_STRIDE, indicating that there is an unpartitioned component before this partition. The stride of this unpartitioned component is set to the value of PI_STRIDE (i.e. S3 is set to S_4), where the value of NS3 is set to $S_6/S_4=(h_4-l_4+1)(h_5-l_5+1)$. The value of PI_STRIDE is then set to S_7 (the MOD of this partition), which is then used to process any further components.

If there is another unpartitioned component remaining then the value of S for this unpartitioned component is set to PI_STRIDE. The value of NS is then set to the total declaration length divided by PI_STRIDE, which gives the number of strides in the high order component. In the current example for T when the

Migration Dimension=1, this means setting S4 to S_7 and setting NS4 to (h_7-l_7+1), where the remaining S and NS parameters are set to 1. The remaining two parameters (ITYPE and MD) are set up as before (Figure 5.39), ensuring that the same parallel set is obtained (Figure 5.35).

Processing Migration Dimension=1 (index 6)

```
CALL CAP_DLB_MIGRATE(T(1+(CAP2_LOW-1)*S2), 1+S6, S6, 1+S3, S3, 1, S2,
                     S2, (CAP2_HIGH-CAP2_LOW+1), S4, S6/S4,
                      S7, total_length/S7, 1, 1, 1, 1, 2, 1)
```

Processing Migration Dimension=2 (index 2)

```
CALL CAP_DLB_MIGRATE(T(1+(CAP1_LOW-1)*S6), 1+S2, S2, 1+S3, S3, 1, S2,
                     S4, S6/S4, S6, (CAP1_HIGH-CAP1_LOW+1),
                      S7, total_length/S7, 1, 1, 1, 1, 2, 2)
```

Processing Migration Dimension=3 (index 3)

```
CALL CAP_MIGRATE(T(1+(CAP2_LOW-1)*S2+(CAP1_LOW-1)*S6), 1+S3, S3,
                 1, S2, S2, (CAP2_HIGH-CAP2_LOW+1), S4, S6/S4,
                  S6, (CAP1_HIGH-CAP1_LOW+1), S7, total_length/S7,
                  1, 1, 2, 3)
```

Figure 5.39: Final generated migration calls when T is 1D-mapped.

5.8.4 Constructing The Migration Call When The Data To Be Migrated Is Unpartitioned

When manually constructing the migration calls (Section 4.7.1) the concept of unpartitioned data was not considered, since partitioned and unpartitioned data were identified and treated in the same way. Every variable in the code was examined and the way in which it was partitioned was noted, after which the necessary migration calls were set up using this information.

When converting existing communications into DLB communications the details of any implicit partitions (if existent) need to be known before constructing the migration calls. The implicit partitions must be extracted before generating the migration calls, as it is necessary to determine whether the data is partitioned in any other dimensions. Therefore, in addition to storing information relating to previous partitions, information pertaining to the execution control mask statements generated in previously partitioned dimensions (passes) also need to be stored.

5.9 Updating The Processor Partition Range Limits

The call to CAP_DLB_REASSIGNLOWHIGH (Section 3.9) to update the processor partition range limits is generated after constructing the migration calls for a particular Migration Dimension (inside the actual procedure which generates the migration calls). Additionally, the call to update the internal processor partition range limits (CAP_DLB_NEW2OLD_LIMITS) is also generated before duplicating any overlap communications (after all of the generated migration calls).

5.10 Duplicating Overlap Communications

As well as ensuring that each processor owns the data within their new processor partition range limits, it is also necessary for each processor to own any data in its halo region. Each processor needs to update its halo region using up-to-date values after load migration (Section 4.7.3).

Those overlap communications whose data is always assigned and used before the next redistribution will not be affected by redistribution, since the halo region will be updated using the current partition. Similarly, those overlap communications whose data is assigned and used after redistribution will not be affected by redistribution since the halo region is always updated using the new partition. If an overlap communication is executed before the load is redistributed, and its data is used after redistribution, then it will be affected by redistribution, since the overlap region of the new partition will not have been updated. After redistribution, the halo region on each processor needs to be updated with the values using the new partition (as the data needs to be owned by a different processor).

Section 5.10.1 will describe how to identify those potential overlap communications that may be duplicated and Section 5.10.2 will discuss the criteria used to determine if a communication should be duplicated and how this is done. If an overlap communication cannot be found (processors assign data in

their halo region), then such a communication may need to be constructed, as discussed in Section 5.10.3.

5.10.1 Identifying Potential Overlap Communications To Duplicate

This stage of the automation process involves identifying those potential overlap communications that may need to be duplicated (Section 4.7.3.1). Not all of the different types of communication (Section A.3.3) need to be processed, just those that update the halo region, which mainly consists of Exchange, but also Receive and Send communications. Even those communications that have been converted into DLB communications may need to be duplicated, which is one of the reasons why the communication type needs to be retained when converting communications (Section 5.6.2). Broadcasts do not need to be duplicated since this type of communication ensures that each processor knows the value of the data being communicated (where the current value will already be known on each processor after redistribution).

An overlap communication will need to be duplicated if the communicated data is used after redistribution. Since the load is redistributed at the beginning of the DLB Loop, the DLB Loop head can be considered as the 'redistribution' point and those overlap communications that may be duplicated can be identified as being executed either before, within or after the DLB Loop (Figure 5.40).

Identified overlap communications are executed either:

- *before the DLB Loop (i.e. above the DLB Loop head)*
- *within the DLB Loop*
- *after the DLB Loop*

Figure 5.40: Classification used to identify overlap communications that may potentially need to be duplicated.

The overlap communications that are executed below the DLB Loop will be communicating data that has already been migrated (i.e. using up-to-date values) and so these communications can be ignored. Similarly, those overlap

communications that are within the DLB Loop can be ignored since they will also be using updated values that have just been migrated. In the latter case, the communicated data will either be used by statements in the same iteration, or after the DLB Loop (after the assignment), or the data will be used in the next iteration of the loop (before the assignment). If used in the next iteration then the communication will have been migrated in the CAPTools generated code to execute at the start of the iteration where the overlap region will be updated using the new partition. The only way that a communication's data is used in the next iteration is if the communication was split (Figure 5.41). The second communication can be ignored as a potential for duplication since it is known that the first communication (identical to the second) will be duplicated.

The data used in those overlap communications that are executed before the load is redistributed may be used before or after load redistribution, where this communication will need to be duplicated if used after redistribution. Therefore only those overlap communications that are executed before the DLB Loop head need to be considered, as they may need to be duplicated.

Original code:

```
do i=
   b(i)=
end do
do it=
   do i=
      a(i)=b(i+1)
   end do
   do i=
      b(i)=
   end do
end do
```

CAPTools generated code:

```
do i=
   b(i)=
end do
call cap_exchange(b...)
do it=
   do i=
      a(i)=b(i+1)
   end do
   do i=
      b(i)=
   end do
   call cap_exchange(b...)
end do
```

Figure 5.41: Example illustrating that only the first of the two identical communications need to be considered for duplication.

In Figure 5.25 for example, where the DLB Loop is in SubDLB (which is called from Sub2) any overlap communication that is executed before the DLB Loop head may have to be duplicated. This involves examining those statements between S1_Main and the call to Sub2 in the Main program, every statement in Sub1, those statements between S1_Sub2 and the call to SubDLB in Sub2, and

every statement between S1_SubDLB down to the actual DLB Loop head (which includes checking everything in Sub4).

Identifying those overlap communications that are executed before control reaches the DLB Loop head can be done in two phases. Firstly, by looking for any halo communications in statements that are executed between the start of the DLB Routine and the DLB Loop head, and secondly by looking for any halo communications in statements that are executed between the start of the Main program down through to any calls to the DLB Routine (Figure 5.42).

Phase 1:
Process all statements from the DLB Loop head block to the start node of the DLB Routine. If any statement is a call to another routine then all statements in that called routine are also processed.

Phase 2:
For each routine calling the DLB Routine, process all statements from the call statement to the start node of the caller routine. Then recursively process the callers of the calling routine. Again, if processing a called routine then all statements in that called routine are also processed.

Figure 5.42: The different phases used to identify overlap communications to be duplicated.

The immediate predominator (Section B.4.1) of each block is used to ensure that every possible communication updating the halo region of data used after load migration is processed. Communication requesters of data will have been migrated up the predominator graph where the communication will definitely be executed. Starting with the block containing the DLB Loop head, its predominating block (PREDOM) is examined, where it is known that this predominating block will always be executed and that any overlap communication found in such a block will definitely be processed. All predominating blocks of the DLB Loop head up until the start node of the DLB Routine can be examined by looking at the PREDOM block of each predominator, as illustrated in Figure 5.43 which shows the code for the FINDPREDLBCOMMS procedure. Similarly, the overlap communications in Phase 2 can be processed by examining all the predominators of each statement which call the DLB Routine up until the Main program, where every calling routine is recursively processed along with their callers. Figure 5.44 shows the code for the FINDPREDLBCALLCOMMS procedure in which the blocks are examined in reversed order.

```
PROCEDURE FINDPREDLBCOMMS(ROUTINE,COMMAND,...);
IF ((NOT ROUTINE^.ONROUTE)) THEN
   (* Routine has not been processed - find communications to duplicate *)
   BEGIN
   (* Mark the routine as having been processed so that it's not processed again *)
   ROUTINE^.ONROUTE:=TRUE;
   (* Set the block to the block of the calling command (which is passed in) *)
   DBLOCK:=COMMAND^.BLOCK;
   WHILE DBLOCK <> NIL DO
      BEGIN
      (* Set the command to the top command in that block *)
      DCOMMAND:=DBLOCK^.COMMANDS;
      IF DBLOCK = COMMAND^.BLOCK THEN
         (* In start block - only process statements until start command reached*)
         STOPCOMMAND:=COMMAND
      ELSE
         (* In a predominating block - process entire block *)
         STOPCOMMAND:=NIL;
      (* Look through commands until the last command or the actual calling command *)
      (* is reached *)
      WHILE (DCOMMAND <> NIL) AND (DCOMMAND <> STOPCOMMAND) DO
         BEGIN
         (* Examine this statement to determine whether it should be duplicated *)
         ...
         (* Process next command *)
         DCOMMAND:=DCOMMAND^.NEXT;
         END;
      (* Process predominating block that is always executed before current block*)
      DBLOCK:=DBLOCK^.PREDOM;
      END;
   END;
```

Figure 5.43: Pseudo code used to process all of the predominating blocks of the DLB Loop head block.

```
PROCEDURE FINDPREDLBCALLCOMMS(ROUTINE,...)
(* Look at calling routine *)
CALLS:=ROUTINE^.CALLEDBY;
WHILE CALLS <> NIL DO
   BEGIN
   (* Check the commands in the calling routine - starting from the calling statement *)
   FINDPREDLBCOMMS(CALLS^.REF,CALLS^.COMMAND,...);
   (* Look at the calling routines of this calling routine being processed*)
   FINDPREDLBCALLCOMMS(CALLS^.REF,...);
   (* Process the next calling routine *)
   CALLS:=CALLS^.NEXT;
   END;
```

Figure 5.44: Pseudo code used to recursively process every calling routine and its callers.

In Figure 5.25 for example, Phase 1 would involve processing all of the statements from the DLB Loop head block up to the start node of the DLB Routine. Phase 2 would involve examining the statements from the call to

SubDLB up to the start node of Sub2 (a calling routine) and then recursively processing the callers (i.e. the Main program) from the call statement. Even Sub5 would be examined since it is called from Sub2 (executed before the call to SubDLB). Note that any other routine calling the DLB routine would be processed in the same manner.

The FINDPREDLBCOMMS procedure is only concerned with duplicating those communications that update the halo region, of which there are only a few circumstances under which these can be generated by CAPTools (Figure 5.45). The first instance shows that the halo region can be updated using a simple communication which can then be tested and duplicated if necessary (Section 5.10.2). Secondly, if the statement being examined is a call to another routine or function (for instance, the call to Sub4 in Figure 5.25), then it is possible that a halo region may be updated inside this routine or function. This essentially means that every single statement in the called routine (function) needs to be processed. A recursive call to FINDPREDLBCOMMS is made since any called routines may contain calls to other routines that will also need to be processed, where the Stop Node of the called routine (CALLS^.REF^.STOP) is passed in as the COMMAND parameter in Figure 5.45. The final instance shows that the halo region can be updated within its own DO or IF structure (such as a buffered or pipelined communication), where the contents of the DO or IF block structure is tested exclusively for communications. The DO or IF block may have purposely been generated by CAPTools as part of a communication.

Example 1	***statement is a communication***	call cap_exchange(…)
Example 2	***statement contains a call***	call sub1 or a=func(b,c)+1
Example 3a	***statement is a DO block containing a communication***	do cap_i=… call cap_receive(…) end do
Example 3b	***statement is an IF block containing a communication***	if (…) then call cap_send(…) end if

Example 1

```
IF (DCOMMAND^.LINK <> NIL) AND
   (DCOMMAND^.LINK^.SYMB^.KIND = KEYCALL) AND
   (DCOMMAND^.LINK^.LEFT^.SYMB^.KIND IN
   [KEYSEND,KEYBUFFER,KEYRECEIVE,KEYUNBUFFER,
   KEYEXCHANGE]) THEN
   BEGIN
   (* Check whether this communication needs to be duplicated *)
   DUPLICATE:=DUPLICATECOMM(ROUTINE,COMMCOMMAND,...,
                                    LISTOFDUPLICATEDCOMMANDS);
```

Example 2

```
IF (DCOMMAND^.CALLS <> NIL) THEN
   BEGIN
   (* Look to see whether this command is calling another routine/function *)
   CALLS:=DCOMMAND^.CALLS;
   (* Look at all the calls of this command - will either be nil or *)
   (* the actual command *)
   WHILE (CALLS <> NIL) AND (CALLS^.COMMAND = DCOMMAND) DO
      BEGIN
      (* Process all called routines except communication calls *)
      (* Set up the call path for this command, and add this called routine *)
      (* onto the list (backwards) *)
      IF (CALLS^.COMMAND^.LINK <> NIL) AND
         ( (CALLS^.COMMAND^.LINK^.SYMB^.KIND <> KEYCALL) OR
         (CALLS^.COMMAND^.LINK^.LEFT^.SYMB^.KIND < KEYSEND) OR
         (CALLS^.COMMAND^.LINK^.LEFT^.SYMB^.KIND >
          KEYRECBUFFER) ) THEN
         BEGIN
         (* Now check all the commands in this called routine - starting from *)
         (* the bottom (STOP node)- process the entire routine *)
         FINDPREDLBCOMMS(CALLS^.REF,CALLS^.REF^.STOP,...);
         END;
      CALLS:=CALLS^.NEXT;
      END;
   END;
```

Example 3a

Example 3b

```
IF (    (DCOMMAND^.LINK^.SYMB^.KIND IN [KEYIF,KEYDO]) THEN
   BEGIN
   (* Examine the contents of the Do/If block for communications *)
   CHECKDOIFBLOCK(ROUTINE,DCOMMAND,...);
   END;
```

Figure 5.45: Examination of processed statement in FINDPREDLBCOMMS (instances from which duplicable communications can be identified).

If the statement being processed is either a DO or IF statement, then the statements in its child block are examined in the CHECKDOIFBLOCK procedure. If the child block only contains communications then these are tested (see Section 5.10.2) and the whole DO/IF block structure is duplicated if necessary. If the child block contains any executable statements other than communications, excluding other DO or IF statements which are recursively examined, then there is no need to consider this block and its communications as it is not a predominator of the DLB Loop (Figure 5.46). In this instance the communications are only required by statements within the DO or IF block structure, otherwise they would have been migrated up above the head of the structure (e.g. above the IF statement), as discussed in Section B.9.1.4. Note that CAPTools will not have generated halo communications using an IF ELSE construct, and so this type of structure need not be considered. If the processed statement is a DO structure then every executable block in the loop (including nested blocks) need to be examined.

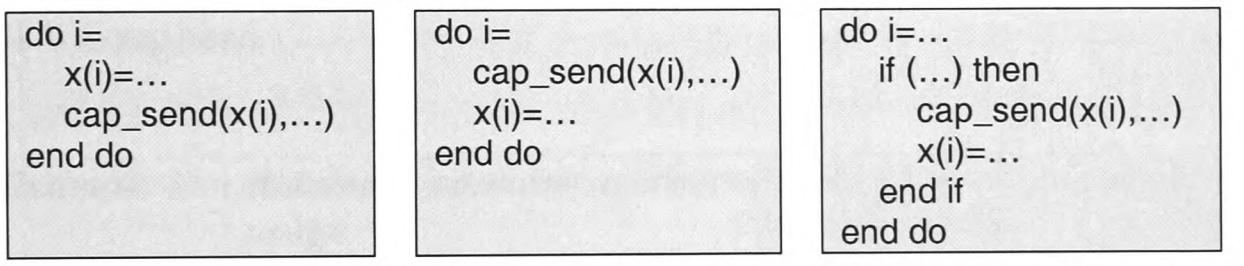


Figure 5.46: Example DO Blocks that contain communication statements and non-communication statements.

5.10.2 Testing The Usage Statements Of The Identified Overlap Communications

Having identified an overlap communication that is executed before redistribution (i.e. before the DLB Loop head), the next stage is to determine whether that communication should be duplicated. A communication only needs to be duplicated if the communicated data is used after redistribution, this means examining all of the usage statements (dependencies) of the identified communication.

Being able to follow the path from assignment to usage (or vice-versa) for a very complex code could be very convoluted. The great advantage in using a tool such as CAPTools to automate this process is that the dependence analysis can be used to view the usage statements of any existing halo communications that need to be duplicated. Since the dependence analysis is interprocedural then this also allows halo communications to be identified from within calling routines.

A communication may have several requesting statements, since the requests of each usage will have been migrated up through the code to execute as early as possible where CAPTools has then merged them (Section B.9.1.5). The usage statement does not always contain a direct usage. For instance, the usage statement of the communication may be a call statement where the communicated data is used inside the called routine (Figure 5.47). Alternatively, the communicated data may be used in another communication statement. Every usage statement needs to be examined, such that the communication will have to be duplicated if at least one of these usage statements is executed after redistribution, which involves comparing the call paths of the usage statement and the DLB Loop head.

Example 1	***statement contains a direct usage***	...=X(...)
Example 2	***usage statement is a call statement***	call sub1 or a=func(b,c)+1
Example 3	***usage statement is a communication statement***	call cap_receive(...)

Figure 5.47: Examples of possible usage statements that require data to be communicated.

The algorithm (DUPLICATECOMM) shown in Figure 5.48 has three components detailed in the following. The call path of the statement containing a direct usage is compared against the call path of the DLB Loop head in order to determine whether the identified communication statement needs to be duplicated after load migration. The DLB Loop head is used as a dummy for the statement that actually redistributes the load, since no other non-DLB statements are executed between the loop head and load redistribution. The call paths will never be the same since the DLB Loop head will be contained in its own block, meaning that the identified communication will only need to be duplicated if the call path

of the usage statement block follows the call path of the DLB Loop head block. For example, consider the situation in Figure 5.25 in which an overlap communication in Sub1 is required by a usage statement in Sub3. The call path of the DLB Routine includes the call to Sub2 in the Main program, which is compared against the call to Sub3 in the Main program. The overlap communication must be duplicated since the call to Sub3 is executed after executing the call to Sub2 that calls the DLB Routine.

If the usage statement of the identified communication is a call statement, then the direct usage statement within the called routine is recursively traced interprocedurally (storing the call path of the direct usage). The direct usage statement is then processed as above.

If the usage statement of the identified communication is another communication statement then that usage communication is recursively processed. If the usage communication has already been processed then the decision to duplicate the identified communication is inherited from its usage communication (i.e. if the usage communication has already been duplicated then the identified communication will also need to be duplicated). It is likely that such usage communication statements will have already been processed if they are not in the same block as the identified communication, since the statements are processed in reverse order using the immediate predominator (Section 5.10.1).

Every communication processed in the DUPLICATECOMM function (Figure 5.48) is stored in a LISTOFDUPLICATEDCOMMANDS along with its call path, where a flag is used to indicate whether the listed communication has been duplicated. In Figure 5.49, the communication in statement S_8 is the first potential communication to be processed (since it is the first communication predominating the DLB Loop head block). The communicated data in statement S_8 (V) is used in statement S_{14} (after the load is redistributed), and so this communication will be stored in the list as having been duplicated. The communication and its surrounding DO construct will then be duplicated on return from this function. The second potential communication to be duplicated is identified on statement S_4 (since S_4, S_5 and S_6 are in the same block), where its communicated data (n_T) is used in statement S_5 which is a communication that has not been processed. The communication in S_5 is then processed recursively in a call to the DUPLICATECOMM function, where it is found that its usage

statement S_{10} is executed before load redistribution. It is therefore decided that the communications in both statement S_5 and S_4 will not be duplicated and this decision is stored in the list. The fourth communication to be identified is that found in statement S_6, whose usage of n_V in S_8 is a communication that has already been processed. As it has already been deemed necessary to duplicate the communication in S_8, then this decision is inherited by the communication in S_6 (where the decision for this statement is also stored in the list). The next communication to be processed is in statement S_1, where it is found to have a direct usage which is executed after redistribution, therefore it will be duplicated. Finally, the communication in statement S_2 is processed, where it is found to have two usage statements (S_3 and S_{15}). The variable being communicated in S_2 (W) is used inside the call to SubA and so the communication will be duplicated.

```
FUNCTION DUPLICATECOMM(CROUTINE,CCOMMAND,...)
(* Examine all usage statements of identified communication statement *)
(* Set DUPLIC to false for this instance of this function *)
DUPLIC:=FALSE;
(* Get dependencies of communicated data *)
CDEPEND:=GETCHILDDEPEND(CCOMMAND,SYMBOL);
WHILE (NOT DUPLIC) AND (CDEPEND <> NIL) AND (CDEPEND^.SYMB = SYMBOL) DO
   BEGIN
   (* Examine dependencies of communication statement *)
   IF (CDEPEND^.DEPENDCOM^.LINK <> NIL) AND
         ((CDEPEND^.DTYPE IN [0,3]) OR (CDEPEND^.DTYPE = -1)) THEN
      BEGIN
      (* True dependence *)
      IF ((CDEPEND^.DEPENDCOM^.LINK^.SYMB^.KIND = KEYCALL) AND
          (NOT CDEPEND^.DEPENDCOM^.LINK^.LEFT^.SYMB^.KIND IN
                                                [KEYSEND..KEYRECBUFFER])) THEN
         (* Usage command is a call to another routine, but not a communication call *)
         BEGIN
         (* Find called routine *)
         CALLS:=CDEPEND^.DEPENDCOM^.CALLS;
         (* Loop over all calls or until duplication has been proven necessary *)
         WHILE (NOT DUPLIC) AND (CALLS <> NIL) AND
                     (CALLS^.COMMAND = CDEPEND^.DEPENDCOM) DO
            BEGIN
            (* Find the name of the variable in the called routine *)
            OTHERNAME(SYMBOL,CALLS^.TREE,NIL,CALLS^.COMMAND,CROUTINE,
                           CALLS^.REF,CALLEDTREE,ARGUSE);
            IF CALLEDTREE <> NIL THEN
               (* Symbol identified in called routine is used *)
               BEGIN
               (* Store call path of this usage command *)
               DUPLIC:=DUPLICATECOMM(CALLS^.REF,CALLS^.REF^.START,...);
               END;
            CALLS:=CALLS^.NEXT;
```

```
        END;
      END
    ELSE IF (CDEPEND^.DEPENDCOM^.LINK^.SYMB^.KIND = KEYCALL) AND
      (CDEPEND^.DEPENDCOM^.LINK^.LEFT^.SYMB^.KIND IN
      [KEYSEND..KEYRECBUFFER]) THEN
      (* Usage command is a communication call *)
      BEGIN
      (* Check whether usage communication is already *)
      (* in LISTOFDUPLICATEDCOMMANDS *)
      IF FOUNDINLISTOFCOMMANDS THEN
        (* Have found a match – inherit decision as to the duplication of the processed *)
        (* command *)
        DUPLIC:=LISTOFCOMMANDS^.DUPLICATED
      ELSE
        (* No match found - find whether communication containing usage is used *)
        (* within or below the DLB Loop *)
        BEGIN
        (* Store call path of this usage command *)
        DUPLIC:=DUPLICATECOMM(CROUTINE,CDEPEND^.DEPENDCOM,...);
        END;
      END
    ELSE
      BEGIN
      (* Direct usage of identified communication *)
      (* Process this command which is not a call to another routine/function or a *)
      (* communication *)
      (* Compare all the call paths of the usage statement block against the call path of the *)
      (* DLB Loop head block (i.e. traverse up the call paths to a common caller *)
      IF FOUNDCALLPATH THEN
        (* Need to find out whether usage command is above the DLB Loop head block *)
        (* (or the call to the DLB Routine), in order to test whether the usage statement *)
        (* is executed before or after the DLB Loop *)
        (* Have matched routines – now search control flow graph - look at the blocks *)
        DUPLIC:=FOLLOWER(DLBBLOCK,USAGEBLOCK,...)
      END;
    END;
  (* Look at next dependence *)
  CDEPEND:=CDEPEND^.NEXTCHILD;
  END;
DUPLICATECOMM:=DUPLIC;
```

Figure 5.48: Pseudo code used to determine whether an identified communication needs to be duplicated.

S_1	**comm(X)**	*process this communication 5th*	
S_2	**comm(W)**	*process this communication 6th*	
S_3	...=W		*direct usage of S_2*
S_4	**comm(n_T)**	*process this communication 2nd*	
S_5	**comm(T,n_T)**	*process this communication 3rd*	*usage of S_4*
S_6	**comm(n_V)**	*process this communication 4th*	
S_7	do		
S_8	**comm(V,n_V)**	*process this communication 1st*	*usage of S_6*
S_9	end do		
S_{10}	...=T		*direct usage of S_5*
S_{11}	do DLB Loop		
	redistribution		
S_{12}	...=X		*direct usage of S_1*
S_{13}	end DLB Loop		
S_{14}	...=V		*direct usage of S_8*
S_{15}	call SubA(W)		*usage of S_2*

Figure 5.49: Example illustrating that the decision to duplicate an identified communication can be inherited by predominating communications.

To retain the original execution order of the duplicated communications (Section A.3.1) the order in which any identified communication is duplicated must be considered. The communications of newly processed blocks must be placed above the duplicated communications of a previously processed block, whereas if processing a block containing several communications then these communications need to be executed in their original order.

5.10.3 New Communications For Assigned Overlaps

It is possible that a processor may have assigned the data within their halo region (Figure 4.22), in which circumstance CAPTools would not have generated any overlap communications for this data. After load migration however, each processor will need to know the values of the data contained in the halo region of

the new partition, which means having to construct and insert new overlap communications for the said data (situated along with any duplicated overlap communications). The constructed communications can be placed anywhere in the duplication section of the DLB code, since their execution does not rely on the execution of other overlap communications. The construction of such communications is currently only undertaken manually since only one code has required this so far, although it is possible to automatically construct these overlap communications by modifying existing CAPTools algorithms which are used to calculate and generate communication calls.

5.11 Results And Observations

This Section reports on the automatic implementation of the DLB Staggered Limit Strategy within several codes including those in which the DLB strategy has been manually implemented within (see Section 4.9).

FAB is a 2D heat diffusion and conduction structured mesh code that was developed in-house at the University of Greenwich. This 670 line code allows for the definition of complex boundary conditions, solving for temperature/enthalpy in two dimensions. The solver is based on the Gauss-Seidal/ Line Successive Over Relaxation (LSOR) algorithm which sweeps the domain in the J-direction solving for each I line.

The APPBT and APPSP codes, 4457 and 3516 lines respectively, are also both part of the NAS benchmark suite [88]. Both codes use an implicit algorithm to compute a finite difference solution to the 3D compressible Navier-Stokes equations, where the solution is based on a Beam-Warming approximate factorisation. The approximate factorisation decouples the three dimensions, leading to three sets of regularly structured systems of linear equations, which are solved as either a system of block tridiagonal equations (APPBT) or scalar pentagdiagonal equations (APPSP).

SWM256 is a 501 line program from the SPEC92 benchmark suite [99]. It performs a two-dimensional stencil computation that applies finite-difference methods to solve shallow water equations.

5.11.1 Overview Of Codes

The following aims to give an overview of the implemented DLB strategy as well as an indication of the involvement of the user. This research has mainly concerned itself with the development of a generic DLB strategy that could be automated within CAPTools. The actual algorithms used to redistribute the workload dynamically can be modified at any time, and so after devising these algorithms most of the effort was focussed on the automatic implementation of the DLB Staggered Limit Strategy.

Table 5.2 gives an overview of the 2D-partitioned application codes in which the DLB Staggered Limit Strategy has been automatically implemented within using CAPTools. The number of serial lines in each code is given along with the variables and indices that have been partitioned. The time taken to parallelise the application (in seconds) on a 700 MHz Pentium 3 processor is shown, where the number of lines in the parallel application is also shown. In the case of SEA (Section 4.9.3), some statements in the code never execute and so they have been dead code eliminated by CAPTools, hence the number of parallel lines is less than the number of serial lines. The number of routines in the parallel code is also shown, which includes any copied routines that CAPTools has generated, since every routine and every communication needs to be examined when implementing the DLB strategy. The total number of communications given for the CAPTools generated parallel code (without DLB) consists of the communications for a 2D partitioning as well as I/O communications. Communications relating to I/O are not listed in the Communications Browser window (Figure B.60), but they may still need to be converted into DLB communications, which is why they are considered here.

The routine selected as containing a significant amount of load imbalance is shown along with the amount of time taken by CAPTools to implement the DLB Staggered Limit Strategy within the given parallel code. Having manually implemented the DLB Staggered Limit Strategy within the JACOBI, APPLU1.4, ARC3D and SEA codes (Section 4.9), the main benefit of automating the process was that the implementation time was reduced dramatically, where the most time consuming aspect of manually implementing the DLB Staggered Limit Strategy

was identifying those communications that needed to be duplicated after redistribution, especially in those codes that involved many routines, which highlights the fact that the automatic implementation of the DLB strategy takes just a few seconds when using CAPTools. Note that the number of additional lines inserted into these codes is not significant as most of the added DLB code consists of the migration calls where the total number of DLB lines is proportional to the number of partitioned arrays in the application.

Table 5.2 also shows the total number of communications that have been converted into DLB communications (Section 3.3), inclusive of those that are considered as offset or special DLB communications. For example, 54 of the 183 communications in APPSP were converted into DLB communications, where 9 of these involve offsets and 30 of these are special DLB communications. Additionally, the number of migration calls in the Non-Staggered and the Staggered Dimensions are given, along with the number of duplicated communications required to implement the new distribution.

In terms of those codes in which the DLB Staggered Limit Strategy has been automatically and manually implemented within, the automatically generated DLB code is the same in appearance as the manually implemented code. The automatically generated code also produces the same results as the manually implemented code, which is evident for example in the self validating APPLU code. An automatically generated DLB parallel version of the FAB code can be seen in Appendix C, where the manual implementation of the DLB Staggered Limit Strategy has not been undertaken in this application code.

	JACOBI	SWM256	FAB	APPLU	APPBT	APPSP	ARC3D	SEA
No. of lines in serial code	37	501	670	3323	4457	3516	4030	7303
1st partitioned variable (index)	tnew (2)	unew (2)	tnew (2)	d (5)	a (5)	d (3)	s (3)	h (1)
2nd partitioned variable (index)	tnew (1)	unew (1)	tnew (1)	d (4)	a (4)	d (2)	s (2)	h (2)
Parallelisation time in seconds	60	120	120	420	780	720	600	1380
No. of lines in parallel code	95	1133	952	4036	4892	5090	6376	6680
No. of routines	1	7	12	19	21	28	26	29
Total no. of communications	18	150	171	120	129	183	601	341
DLB Routine	main	shallow	solver	ssor	badi	adi	main	main
Implementation time in seconds	**<1**	**<1**	**<1**	**1**	**<1**	**<1**	**<1**	**<1**
No. of lines in DLB parallel code	146	1253	1066	4147	5001	5232	6892	6998
Total no. of DLB communications	4	38	4	27	34	54	210	100
- offsets	0	0	0	9	9	9	3	0
- special	2	0	0	3	11	30	162	16
Non-SD migrations calls	2	14	7	7	6	8	25	17
SD migration calls	2	14	12	7	6	8	26	34
Duplicated communications	0	0	12	4	4	4	48	15

Table 5.2: Overview of the automatically dynamically load balanced codes, where the given times are taken from a 700 MHz Pentium 3 processor.

5.12 Summary

This Chapter has illustrated how CAPTools has been extended to automatically implement the DLB Staggered Limit Strategy within a CAPTools generated parallel code using a generic DLB algorithm.

This Chapter has shown how existing communications are converted into DLB communications and how the DLB implementation code is set up given a specified DLB Loop. Issues such as where to insert the DLB implementation code were discussed, formalising the approach used when manually implementing the DLB Staggered Limit Strategy. The construction of the migration calls focused on the data structures available in CAPTools, emphasising the generic nature of the calls. In addition to ensuring that a processor owned the data between its new processor partition range limits, the overlap region of data used after redistribution also had to be updated. Overlap communications had to first be identified and then duplicated under certain conditions. This Chapter has stressed that when using CAPTools with the added functionality of DLB, the user effort required to automatically implement the DLB Staggered Limit Strategy within a parallel code is minimal with the press of a few buttons. Automation also enables research into the details of DLB (when, what, etc) to be examined by altering the algorithms in the devised utilities (Chapter 3) which can be tested on a large number of automatically generated parallel application codes.

This Chapter has demonstrated the usefulness of the DLB Staggered Limit Strategy, particularly the practicality it offers in being a generic strategy that can be automated within CAPTools. The DLB strategy can be applied to a wide range of structured mesh application codes, where it is then possible to improve the algorithms used to determine when to redistribute the workload and how much should be redistributed.

Chapter 6 Automatically Implementing A Dynamic Load Balancing Strategy Within A CAPTools Generated Unstructured Mesh Code

The context of DLB in this research has been concerned with structured mesh codes, where the implementation of the DLB Staggered Limit Strategy within a parallel code has been automated within CAPTools. This Chapter discusses some of the issues that need to be considered for the automation of a DLB strategy within a CAPTools generated unstructured parallel mesh code, comparing the structured and unstructured methods, highlighting the similarities.

6.1 Unstructured Mesh Codes

Unstructured mesh codes can also be used to solve scientific numerical problems, but in this instance the nature of the mesh is not regular (as illustrated in Figure 6.1). The irregularity of the unstructured mesh allows more flexibility than a structured mesh code when constructing the problem using a complex geometry. The main difference compared to using a structured mesh code is the added complexity in constructing such a code and also the higher memory requirement (with the use of pointers), however its popularity is rising with the advances in technology.

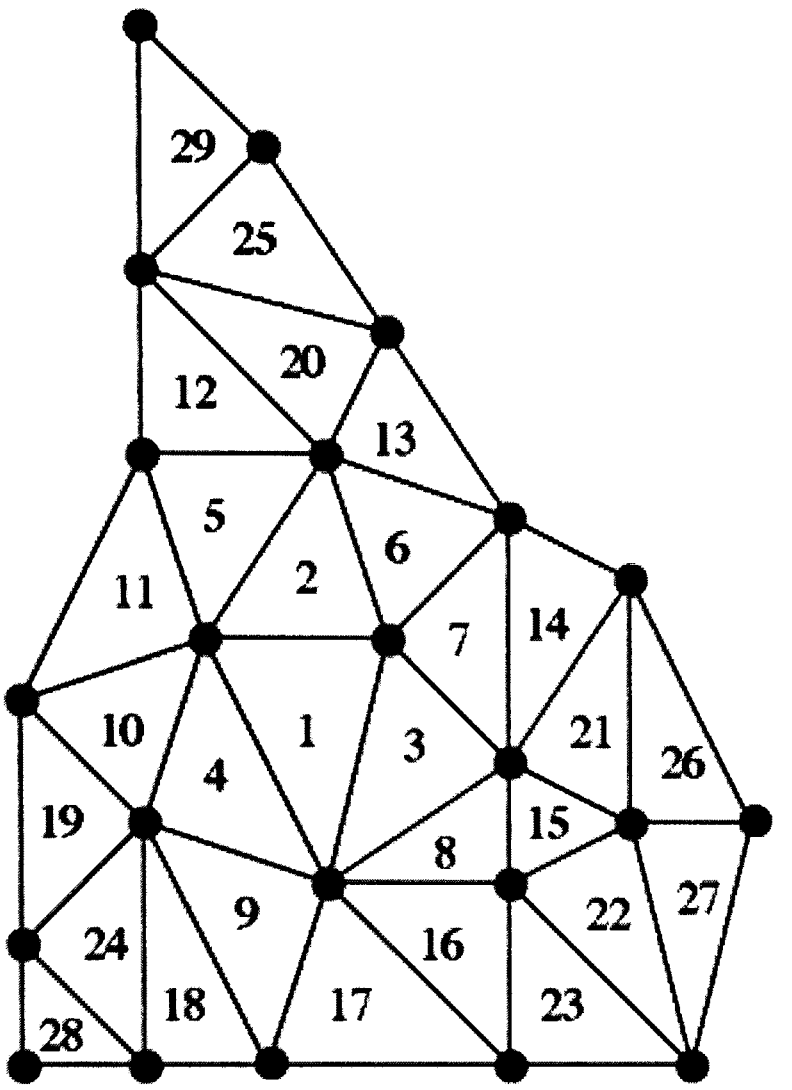


Figure 6.1: Example of an unstructured mesh.

6.2 The Parallelisation Of An Unstructured Mesh Code Using CAPTools

CAPTools can be used to parallelise unstructured mesh codes [100, 101] in a similar manner to that used to parallelise a structured mesh code, allowing the problem to be distributed onto several processors whilst attempting to minimise inter processor communication. After performing a dependence analysis of the application code a partition of faces/elements/nodes is prescribed by the user using the 'Unstructured' option in the Partitioner Browser window (Figure B.35). Execution control masks and communications can then be generated, where the parallel version of the unstructured mesh code will need to be compiled with the '*–unstruct*' switch. Since each processor may have several neighbours, a Full communication topology (Section A.3.2) will need to be employed such that each processor is able to communicate with any other processor (removing the

abstraction of a neighbouring direction). As with the partitioning of structured mesh codes, the core elements (faces or nodes) are owned by processors to which they are allocated, with the halo elements being owned by neighbouring processors.

The application is initially partitioned using a cyclic partition, where this is improved upon with a call to Jostle [61, 63, 64, 102, 103, 104, 105] (a graph partitioning tool) which aims to minimise processor interconnectivity. Alternatively a tool such as Metis [66] could be used. Figure 6.2 shows the distributed mesh after the call to Jostle, where the original global numbering scheme is still used. Note that there is no concept of processor partition range limits with parallel unstructured mesh codes, instead a global processor ownership array (CAP_P) is utilised (returned from Jostle) that relates each element in the mesh to a single processor, which is then used to enforce the execution control masks taking the form:

```
IF (CAP_P(I).EQ.CAP_PROCNUM) A(I)=...
```

where CAP_PROCNUM is the unique processor identification number of the executing processor. This basic 'owner compute' rule (Section B.8) ensures that each processor performs computations relating to owned data only, in which the halo region is updated via message passing communications sent from the owning processor to the using processor.

Jostle can also be used to return a locally numbered mesh (Figure 6.3) when the 'Reduced Memory' option is selected (Section B.10), eliminating the need to store the entire mesh. The side-effect of this is that it allows many loops to be adjusted to only pass over the locally owned set, achieved by changing loop limits to be based upon the number of locally owned elements which enables any execution control masks within those loops to be removed. This minimises changes to the user's original serial code, allowing for easy maintenance and optimisation. The elements/nodes owned by a processor are renumbered into their local form in ascending order, whereas halo elements/nodes are renumbered in an arbitrary order. For example, globally numbered element 3 in Figure 6.2 is processed as local element 1 on Processor 3 in Figure 6.3. The local processor ownership array (CAP_P) is used in conjunction with a LOC2GLO array which is

used to convert a locally numbered element back into a globally numbered element when required.

The idea of communicating data in an unstructured mesh application is different to communicating data in a structured mesh application. Inspector loops that convert the data structures used in the application code into a structure understood by the communication library CAPLib (and also by Jostle) are used since such application codes will not all be written using the same data structures.

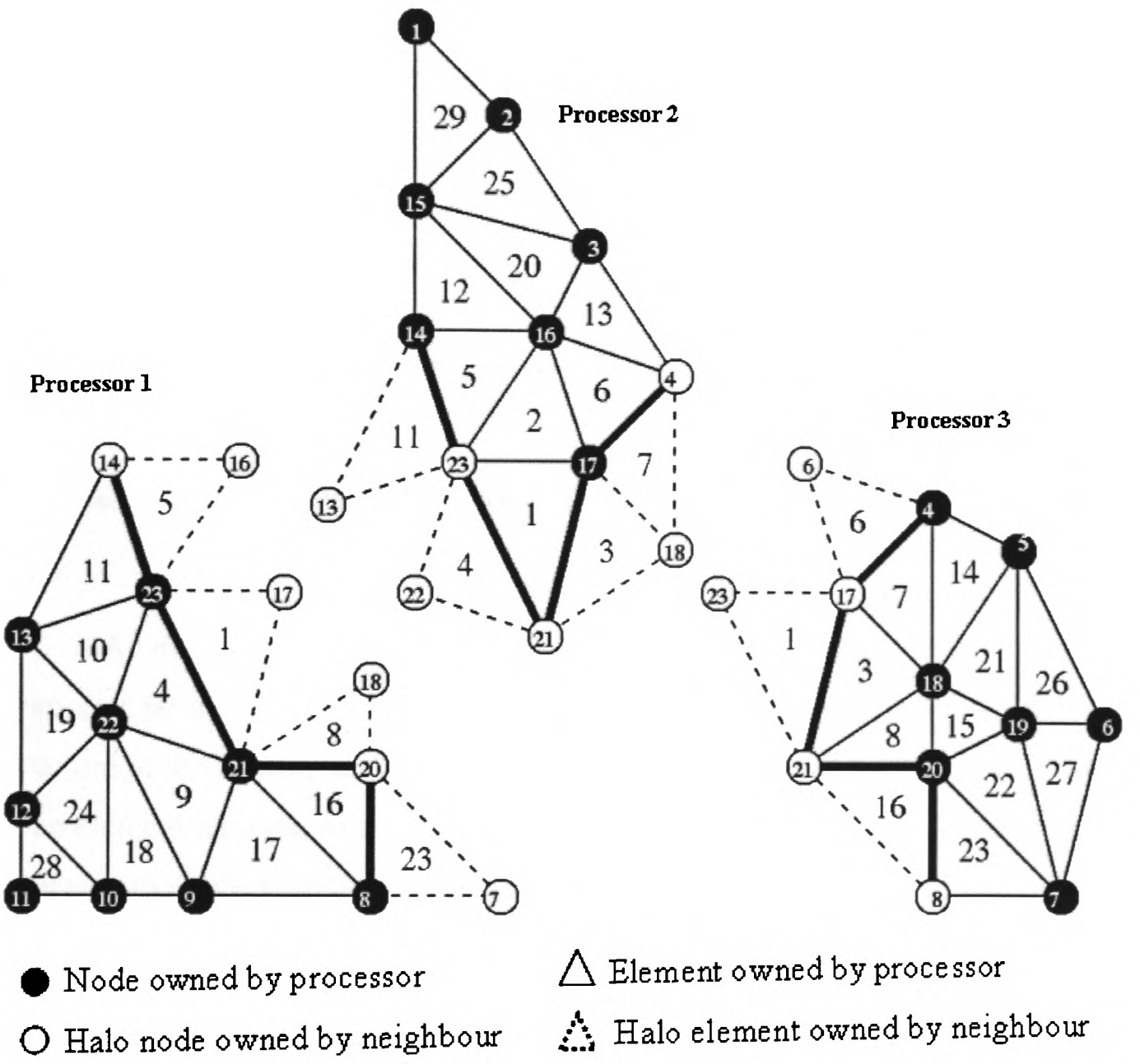


Figure 6.2: Example of the unstructured mesh in Figure 6.1 that has been partitioned onto 3 processors, where global numbering is used.

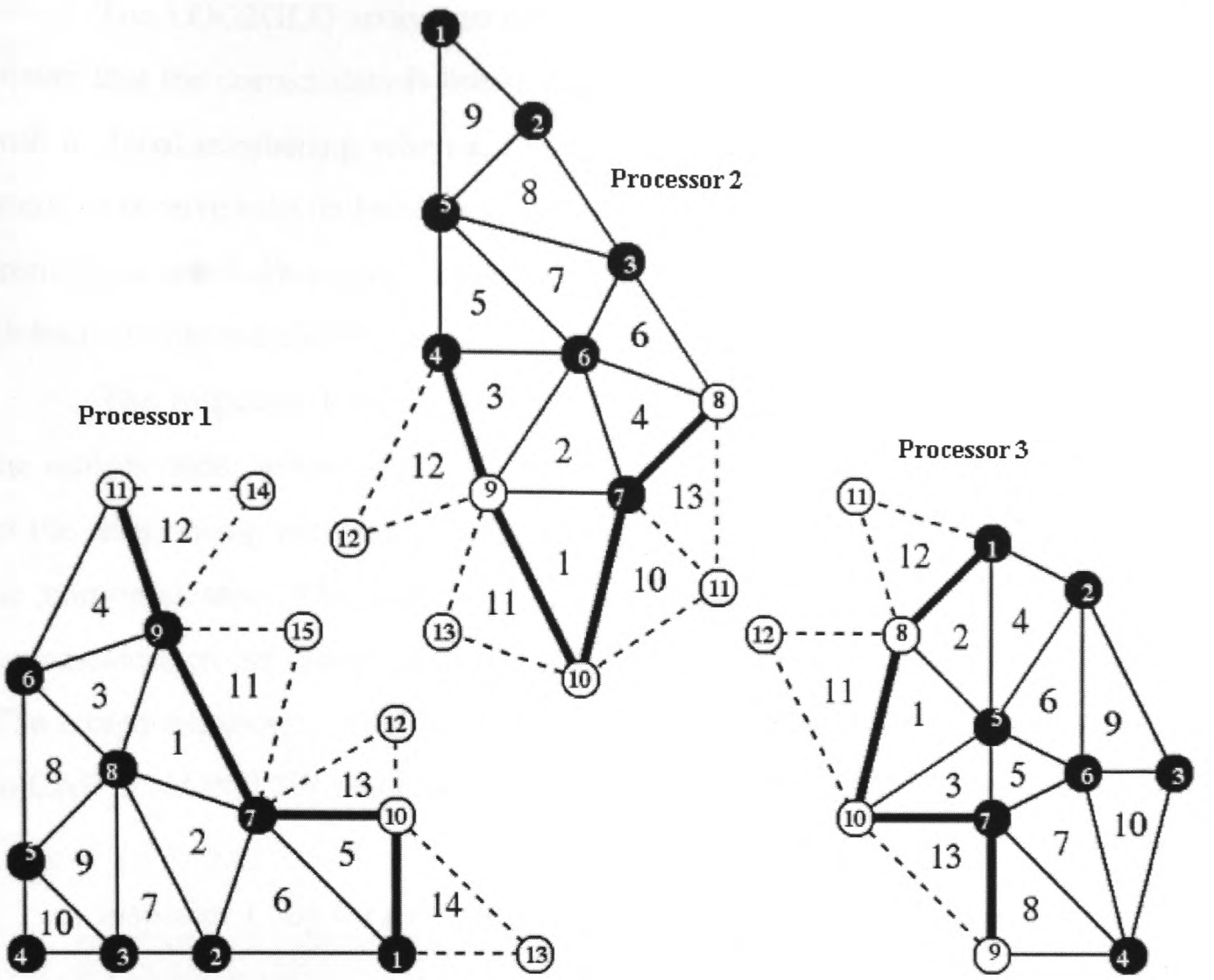


Figure 6.3: The partitioned unstructured mesh (shown in Figure 6.2) with local numbering used.

An example of an inspector loop and the sample code that it was generated from can be seen in Figure 6.4 where local numbering is employed. The data structure of the unstructured mesh is stored in MESH, where the first index relates to an element number, and the second index relates to the j^{th} node of that element, returning the node numbers relating to that element. In Figure 6.3 for example, the locally numbered element 1 on Processor 3 consists of 3 nodes, which are stored as MESH(1,1)=5, MESH(1,2)=8, and MESH(1,3)=10. The sample code in Figure 6.4 evaluates the temperature for each element in the mesh (TEMPELE), based on the average temperature of the surrounding nodes for that element (TEMPNODE). In the case of element 1 on Processor 3, this means averaging the temperature of nodes 5, 8, and 10. However, the temperature for node 8 is stored on Processor 2, and the temperature of node 10 is stored on Processor 1. This means that the temperature of node 8 (locally stored as node 7 on Processor 2) and the temperature of node 10 (locally stored as node 7 on Processor 1) will need to be received from the respective neighbours.

The LOC2GLO array and the local processor ownership array are used to ensure that the correct data is transferred, since communications can only operate with a global numbering scheme. For example, Processor 3 does not know that it needs to receive into its local element 10 the value of locally numbered element 7 from Processor 2. However, Processor 3 can determine that it needs to receive into globally numbered element 17 which is owned by Processor 2.

The inspector loop is constructed from the loops and masks surrounding the sample code, where a call to CAP_CONNECT sets up the connectivity graph of the relationship between I and MESH(I,J) to enable values of TEMPNODE to be communicated. The call to CAP_OVERLAP is then used to construct a communication set based on a calculated receive set and its associated send set. The communication set (indicated by the CAP_ID variable) is then used in the call to CAP_SWAPOVER that communicates values of TEMPNODE before required.

Inspector Loop for the code below:

```
      DO I=1,LOC_NUMELE
        DO J=1,3
          CALL CAP_CONNECT(I,MESH(I,J))
        END DO
      END DO
      CALL CAP_OVERLAP(CAP_P_ELE,CAP_NODE,...,CAP_ID)
```

Sample code:

```
      INTEGER MESH(MAXELE,MAX_NODES_PER_ELE)
      REAL TEMPELE(MAXELE),TEMPNODE(MAXNODES)

C     mesh(element,number_of_nodes_for_element)=j^th_node_of_element

      CALL CAP_SWAPOVER(TEMPNODE,...,CAP_ID)
      DO I=1,LOC_NUMELE
C       Process local elements
        TEMPELE(I)=0.0
        DO J=1,3
C         Sum the temperature of the j^th node for processor I
          TEMPELE(I)=TEMPELE(I)+TEMPNODE(MESH(I,J))
        END DO
        TEMPELE(I)=TEMPELE(I)/3.0
      END DO
```

Figure 6.4: Sample code and the inspector loop used to set up the communication set needed to update data in the halo region in which a local numbering scheme has been used.

The manual parallelisation of a 2D unstructured mesh code called UIFS (Unstructured Incompressible Flow and Stress) [106] was undertaken at the University of Greenwich. The code was developed to model the processes

involved in metals casting, solving the Navier Stokes equations for either transient or steady state flow problems with solidification, along with elastic stress-strain equations [107, 108]. It took over one year to manually produce the parallel version of the code (PUIFS) [7], where the majority of the process was fairly straightforward, albeit very time consuming. Subsequently, CAPTools has been developed based on this experience, and can parallelise the UIFS code in a few hours [100, 109].

6.3 Load Imbalance Within An Unstructured Mesh Code

The problems of load imbalance (discussed in Chapter 1) also exists within unstructured mesh codes since similar assumptions are made during the parallelisation process as were made during the parallelisation of structured mesh codes. It is assumed that each cell will take the same amount of time to compute and that there will be no variation between processor speeds (or number of jobs/users). Once again, this assumption is not always correct as there may well be some variation between processors when using a heterogeneous system of processors, or the computational load may vary due to the physical characteristics of the application code. As with structured mesh codes, load imbalance can be classified as either processor or physical imbalance.

6.4 Dynamic Load Balancing

Much previous research in DLB for unstructured mesh codes is discussed in Section 1.12. Redistributing the workload can reduce the maximum processor iteration time, where the load on the slower/heavily loaded processors is decreased, and is conversely increased on the faster/lightly loaded processors. The same reasons given for structured mesh codes can be used to justify the necessity for dynamic load balancing, for which the goals are the same (Section 2.1). The process has the same stages as for structured mesh codes [87]. The

communications are based on the communication sets, so if the communication sets are recalculated after redistribution, then the communication calls are unchanged.

6.4.1 Where To Redistribute The Workload

As with structured mesh codes, DLB of unstructured mesh codes should be carried out when load imbalance is suspected of causing parallel inefficiencies, placing the code that redistributes the load within a loop containing the load imbalance. The loop containing the load imbalance can be identified in the same manner as with structured mesh codes, for example using a profiler or user knowledge to select a loop that may contain a significant amount of load imbalance. Therefore, in this context there is no extra effort involved in dynamically load balancing an unstructured mesh code.

6.4.2 Determine When To Redistribute

The decision of when the load should be redistributed is independent of the type of mesh. Therefore, as with structured mesh codes, the model of computation (Section 2.7.2.2) can be used. As this decision is evaluated every iteration, the calculation should be quick and simple in order to minimise any overheads involved in the calculation. Again, there is no additional effort required at this stage in terms of automation, since the same calls are placed in the user's code as were used with structured mesh codes.

6.4.3 Calculating The New Distribution

A call can be placed in the user's code which will calculate the new distribution. The new workload on each processor can be calculated using a call to Jostle [65] passing in a weight array relating to the computation time that will determine which cells need to be redistributed, and onto which processors these cells need to be migrated. Jostle can handle both processor and physical imbalance. Unlike structured mesh codes where the calculation of the new workload was restricted by the structure of the mesh, Jostle incorporates the fact that single cells can be shifted onto neighbouring processors, making the algorithm more effective (and less prone to load oscillations). Calculating the new workload is more flexible when dynamically load balancing an unstructured mesh code. Jostle attempts to minimise the number of edge cuts and this in turn attempts to minimise communications with neighbouring processors, whilst obtaining a flexible load balance. Jostle also needs to ensure that the new distribution will operate correctly even if the user has selected the 'Reduce Memory' option where a restriction is placed on the number of gained cells.

6.4.4 Implementing The New Distribution

Having calculated the new distribution using Jostle, the load needs to be migrated in order to implement this distribution [87] (Section 2.8). Generic utilities can be used to migrate the load between processors, minimising the changes to the user's code and hiding the underlying operations from the user. The core set of elements (and related faces and edges etc) are updated using CAP_SWAPCORE to ensure that each processor owns those elements that it operates on. Note that this utility is very similar to the DLB migration utilities discussed in Section 3.7. Elements (faces and edges etc) on the boundary also need to be moved onto other processors, which can be achieved using CAP_SWAPOVER (comparable with the duplication phase discussed in Section 4.7.3). These migration calls can be placed immediately after calculating the new workload, confining the changes to the user's code to just a small section of the parallel code. The communication

sets need to be modified to account for the new distribution, after which the halo region must be updated.

The process of implementing the new distribution is essentially the same as that used with structured mesh codes. Identifying those arrays that need to be migrated (Section 5.8) and identifying those communications that need to be duplicated (Section 5.10) can use identical algorithms as were developed for structured mesh codes. Unlike structured mesh codes however, the pointer arrays (such as MESH in Figure 6.4) need to be renumbered using an inspector loop before continuing with execution if employing reduced memory.

6.5 Summary

The problem of load imbalance is not exclusive to parallel structured mesh applications, it also affects the performance of parallel unstructured mesh codes. Many of the strategies used for structured mesh applications can be applied to the implementation of DLB within a parallel unstructured code. The algorithms used to decide where to redistribute and how often to redistribute are the same, whereas the calculation and implementation of the new distribution follow similar ideas. In terms of automating such a DLB strategy for unstructured mesh applications, the process is very similar, mainly just changing the names of the called utilities.

Chapter 7 Conclusions And Further Work

This research has shown that the automatic implementation of the DLB Staggered Limit Strategy can lead to an increase in parallel efficiency as well as dramatically minimising the user effort required to produce a DLB parallel version of a serial structured mesh application code. The automation of this DLB strategy within CAPTools (Chapter 5) allows the user to quickly and easily implement a DLB parallel code with the press of a few buttons, enabling the user to spend their time obtaining results instead of concentrating on implementing DLB within their code.

This research has focused on the detrimental effects of load imbalance on the parallel performance of structured mesh application codes. Issues surrounding DLB were discussed with the aim of improving the utilisation of the available hardware, such as deciding the location at which to redistribute the load, and how often the load should be redistributed. A generic DLB strategy was devised based on a CAPTools generated parallel code, as one of the aims of this research was to automate the implementation of the devised strategy within CAPTools.

Several DLB strategies were discussed, where it was decided that the DLB Staggered Limit Strategy would be implemented in this research, using coincidental (global) processor partition range limits in all but the last partitioned dimension where non-coincidental (local) limits are used. Due to the flexibility of the staggered limits, the DLB Staggered Limit Strategy offers a reasonably good load balance in comparison to using a strategy which utilises global processor partition range limits in every dimension. The DLB Staggered Limit Strategy is relatively straightforward to construct as, unlike the strategy which utilises local processor partition range limits in every dimension, it retains the rectangular partitions employed by CAPTools, resulting in fairly neat and simple communication patterns without major changes to the user's code. However, one attribute of the DLB Staggered Limit Strategy was that although communications in the Staggered Dimension remained with immediate neighbours, a processor may have to communicate with several neighbours when communicating in a Non-Staggered Dimension across the staggered limits.

Generic utilities were devised for the DLB Staggered Limit Strategy which would be capable of handling multi-dimensional partitioning as well as the staggered limits whilst keeping the underlying operations hidden from the user.

Assuming either processor or physical load imbalance, the utility determining the new processor workloads returned the new processor partition range limits, where these limits were used in the devised DLB communication and migration utilities. Using these main utilities and the other utilities discussed in this thesis, the DLB Staggered Limit Strategy was implemented manually within several codes. Following the procedures and experience gained in manually implementing this strategy, algorithms to automate the DLB Staggered Limit Strategy using CAPTools were formulated and tested.

7.1 Additional Functionality And Future Improvements

The aim of any parallelisation is to efficiently utilise the available hardware whilst obtaining a good quality parallel performance. Whilst the devised DLB Staggered Limit Strategy provides one method of combating the possible effects of load imbalance within a structured mesh application code, there is still room for improvement. These improvements can also be applied to the automatic implementation of DLB within an unstructured parallel application code generated by CAPTools.

The algorithm that determines when to redistribute the workload between processors can be improved to take both processor and physical imbalance into account to cater for the situation in which both types of load imbalance are present. Automatic detection of the type of load imbalance may be useful in reducing load oscillation, possibly detecting those situations in which the load imbalance is changing continuously throughout execution. Similar improvements can be made to the algorithm that determines the new workload distribution, possibly implementing and testing alternative algorithms, as well as making it an iterative process using the timings of an iteration to estimate the timings of the next iteration given the new distribution. It may even be possible to perform this algorithm in parallel [45].

This DLB strategy currently relies on the user to determine where to redistribute the load by selecting a loop containing a significant level of load imbalance, but this selection could be automated (possibly with the aide of a code profiler) such that the main body of load balancing code is placed at several locations containing load imbalance throughout the application code. The model of computation determining the iteration at which to redistribute the load would then automatically determine at which location the load needed to be balanced.

The Information Power Grid [110] is one example in which the application of DLB would be beneficial. Large computing facilities around the world have grouped their processing capabilities together to offer the use of a potential super computer. However, although the proposed processing power seems superior, the issues of load imbalance still exist. If the speed of one processor in the Grid is extremely slow then this will affect the overall parallel efficiency. In future, the algorithm that determines the new workloads could take into account the processor speeds, memory size and communication costs so that the Grid can be utilised efficiently.

7.2 Final Remarks

The issue of DLB with structured mesh codes and its automatic implementation within CAPTools has proven to be a very interesting field of research. The fact that a DLB strategy can be automatically implemented within a structured mesh parallel code using CAPTools enables further research into the investigation of improving the algorithms used for this research along with sensitivity analysis and possibly implementing some of the techniques published by other groups to allow comparisons.

Many application codes are neither computationally balanced nor executed exclusively on a homogeneous system of processors, most involve some form of adaptivity (with the application or with the machines utilised). Parallelising an application code is difficult enough without having to consider the implementation of a DLB strategy. This research makes it possible to automatically implement a DLB strategy within a parallel structured mesh code

that has been parallelised using CAPTools where the user effort required to implement such a strategy is reduced to the press of a few buttons, allowing the user to concentrate their efforts elsewhere.

I believe that this research offers the user a great deal of control over their DLB application code such that the generated code is relatively easy to understand due to the transparency of the devised utilities, and because the user need not write their code using the data structures of some sort of DLB system. The generic nature of the devised DLB Staggered Limit Strategy allows the strategy to be implemented within many real world application codes parallelised by CAPTools and not just on a single application code.

Appendix A The CAPTools Parallelisation Strategy And Communication Library

Chapter 1 discussed the issues surrounding parallel processing and the problem of parallel inefficiencies caused by load imbalance. Having decided that the DLB strategy should be automated within CAPTools, this Appendix aims to explain the fundamentals of CAPTools that are used to attain a scalable, efficient, parallel code using this parallelisation tool which is comparable to a manually parallelised code.

The basic goals of parallelisation were discussed in Section 1.6, where the main objectives of CAPTools were discussed in Section 1.8. A basic understanding of the underlying foundation of CAPTools is vital in order to comprehend the DLB strategy discussed in Chapter 2 and subsequent Chapters. This Appendix discusses the parallelisation strategy used by CAPTools to produce efficient parallel code, where the algorithms and data structures to do this are explained in detail in Appendix B.

A.1 What Is CAPTools?

CAPTools is a semi-automatic parallelisation tool that allows the user to interactively generate a parallel version of their serial Fortran 77 code using the best manual parallelisation techniques. Minimal changes are made to the user's code to avoid alteration to the original algorithm, enabling the user to recognise and easily maintain and optimise their code. Any alterations to the existing algorithm may lead to incorrect results or a degradation of the convergence due to a change in the execution order [111], and so CAPTools must not change the algorithm (although the user may explicitly instruct CAPTools to ignore a dependence). Using the SPMD paradigm, the parallel code should be able to execute on a number of different machines types with different processor

configurations, where communications are kept to a minimum to achieve reasonable speed-up.

Initially focussing its attention on scientific numerical codes, such as structured mesh based Fortran codes, it can be used to parallelise a wide range of application codes. Computational Fluid Dynamics, Heat Transfer and Structural Analysis problems are examples of the main types of problem being parallelised using CAPTools.

The main aim of CAPTools is to produce a parallel code that complies with all of the requirements of parallel processing, which are outlined in Section 1.6. One of the main benefits of using CAPTools is that the parallelisation time of a code can be reduced from weeks or months, to days or even hours, meaning that the user need not spend an unnecessarily long amount of time in the parallelisation stage.

Being able to parallelise a code quickly (and with confidence) using a tool such as CAPTools enables the user to concentrate on the deeper aspects of their problem, such as trying to improve their algorithm rather than improving their parallel code (which they may not be qualified to do). CAPTools also makes it possible for non-expert users to parallelise codes, as it is no longer necessary to have been involved with the development of the code in order to parallelise it, so long as the user has a basic understanding of the code.

CAPTools aims to produce efficient parallel code with good memory usage. Interaction with the user is vital in producing such a code, and so CAPTools uses a Graphical User Interface (GUI), where the main CAPTools window can be seen in Figure A.1 to extract user knowledge and almost automatically parallelise the user's code.

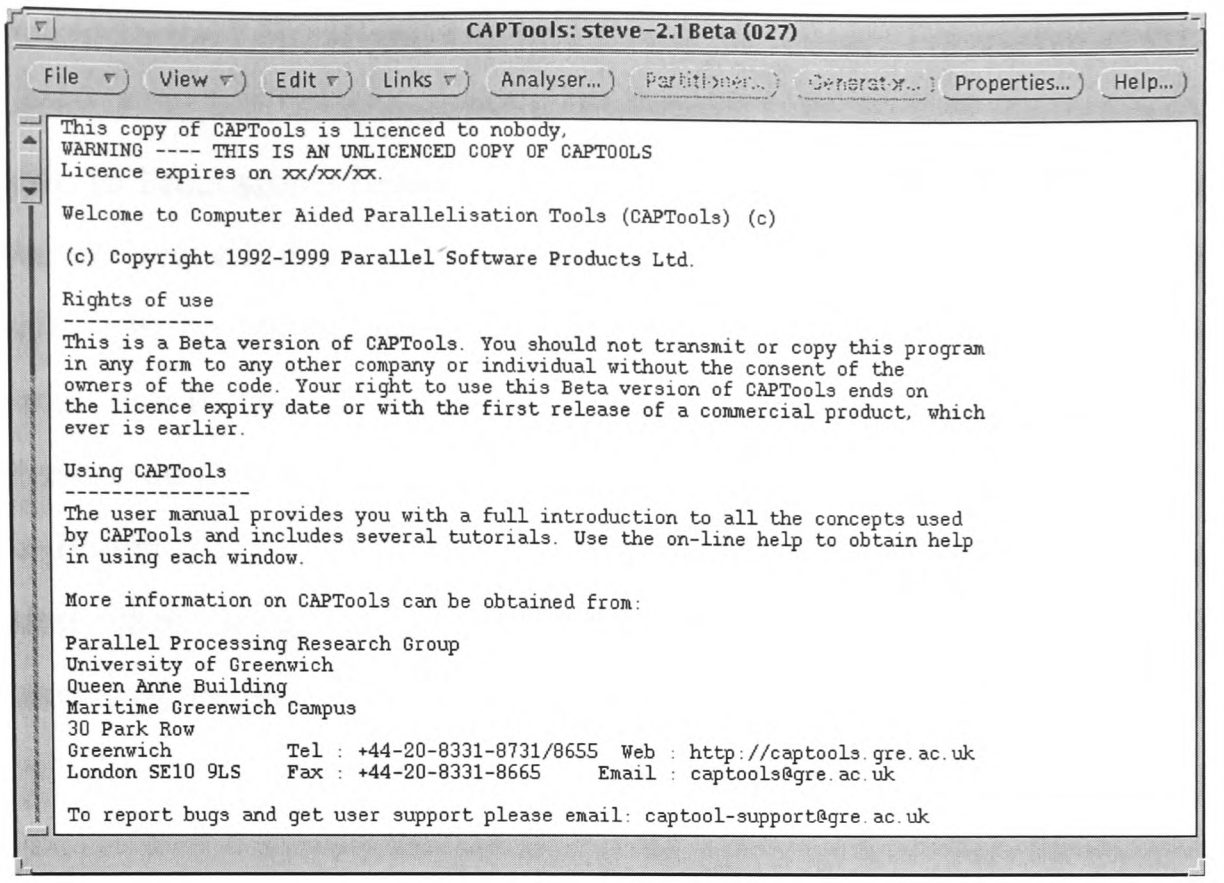


Figure A.1: The main CAPTools GUI window, used to parallelise serial Fortran 77 codes.

A.2 The Parallelisation Of Structured Mesh Codes

As stated earlier, CAPTools is a semi-automatic tool that enables the user to partition several dimensions, generated one at a time, giving the user flexibility over how to distribute the data. Each partitioned dimension can also be referred to by the respective processor axis (IAXES), where processor axis 1 refers to the first partitioned dimension, processor axis 2 refers to the second partitioned dimension, and so on.

The processor configuration (topology) is specified by the user at runtime, indicating the total number of processors used (CAP_NPROC) in the parallel execution of a code. The user specifies how many processors are required using either a Pipe, Ring, Grid, Torus, or Full topology, as demonstrated in Figure A.2. A Pipe configuration (Figure A.2a) is used to connect a line of neighbouring processors, whereas a Ring configuration (Figure A.2b) is an extension of this in which the first processor is connected to the last processor (i.e. Processor 1 and Processor 5 are connected). A Grid configuration is usually used when a 2D partition has been implemented, where each processor has up to four connecting

neighbours (in the Left, Right, Up, and Down, direction). For example, Processor 2 in Figure A.2c will be connected to Processor 1 on the Left, to Processor 3 on the Right, to Processor 5 below, and will have no neighbouring processor above. If the user wanted the same connectivity for Processor 2, but also wanted Processor 2 to be connected to Processor 8 then the user would have to use a Torus topology instead (Figure A.2d). Finally, if the user wanted each processor to be connected to every other processor then Full would have to be used. Note that in Figure A.2e the diagram indicates that the processors are in a grid formation when using Full, but a grid has simply been used to illustrate the connectivity of the processors in the topology.

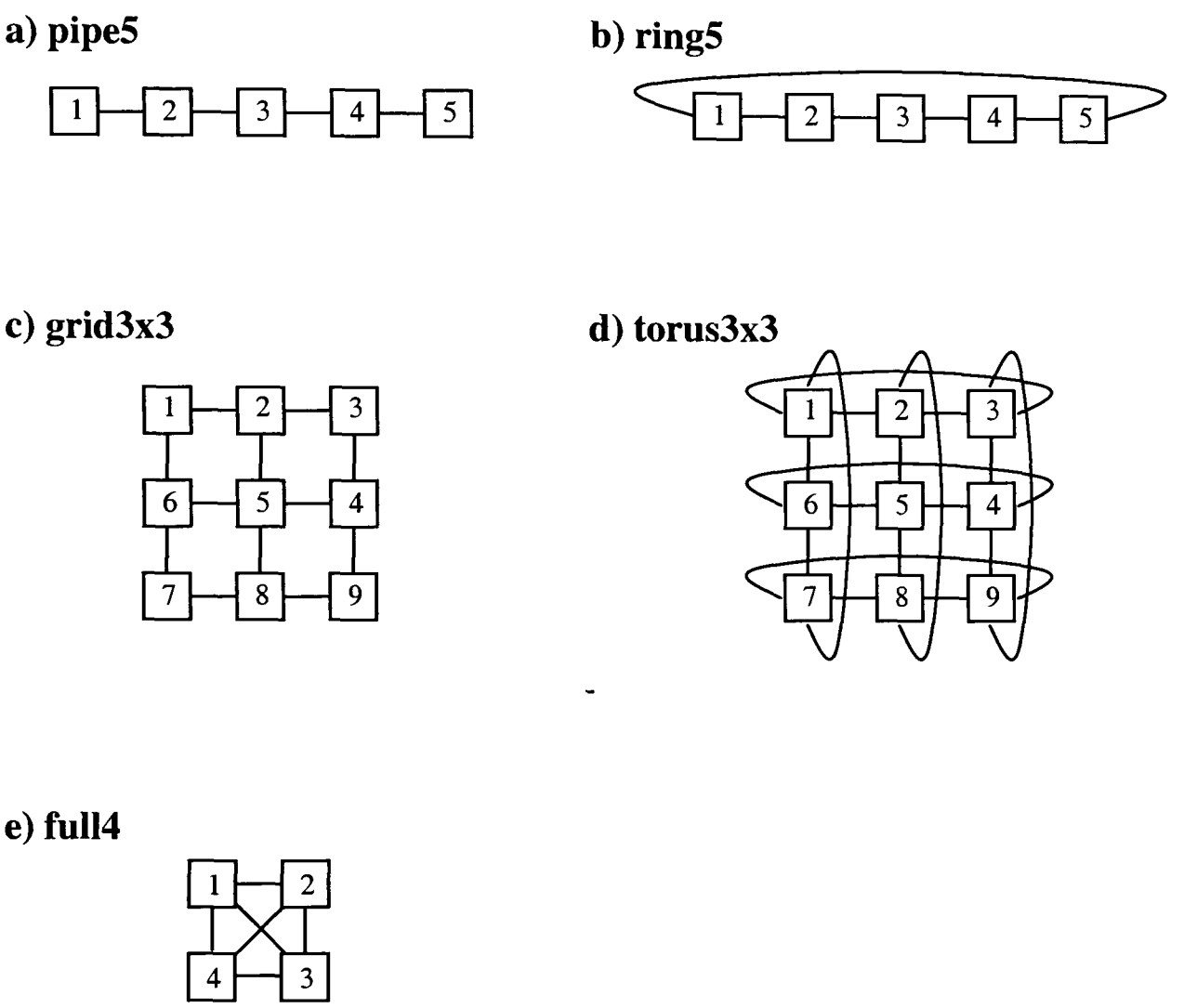


Figure A.2: The different processor topologies used to represent the processor configuration, along with part of the necessary terminology used at runtime to execute the parallel code.

Each processor is uniquely numbered (using CAP_PROCNUM) beginning with Processor 1 (in which a 'snaking' effect is used to number the processors), where each processor can also be identified by its unique position in the topology. The number of processors in a particular partitioned dimension (IAXES) can be

obtained using CAP_DNPROC(IAXES), where the processor position in a particular partitioned dimension can be extracted using CAP_DPROCNUM(IAXES). The values of CAP_NPROC, CAP_DNPROC, and CAP_DPROCNUM, are set up at runtime at the beginning of the parallel code with a call to CAP_INIT. CAP_INIT initialises the code to run in parallel using the specified processor configuration (where the user knows whether a 1D, 2D, or 3D, partition has been used). For example, given a 3D array has been partitioned first in the I direction, then J direction, and finally the K direction (shown in Figure A.3), the user may specify a grid4x3x2 configuration where each processor will know that there are CAP_DNPROC(1)=4 columns of processors, CAP_DNPROC(2)=3 rows of processors, and CAP_DNPROC(3)=2 planes of processors, giving CAP_NPROC=24 processors in total. Processor 6, for example, can also be identified using its position in the topology, where it is found in column CAP_DPROCNUM(1)=3, row CAP_DPROCNUM(2)=2, and plane CAP_DPROCNUM(3)=1. Note that CAP_ is a CAPTools generated variable that has been introduced into the code specifically for running in parallel, where all other CAPTools generated variables follow this format, allowing the user to distinguish between their original code and any CAPTools inserted code. Alternatively the user could have specified a grid6x4x2 configuration (the choice is theirs at runtime), where the CAP_ variables would be set up in the same way.

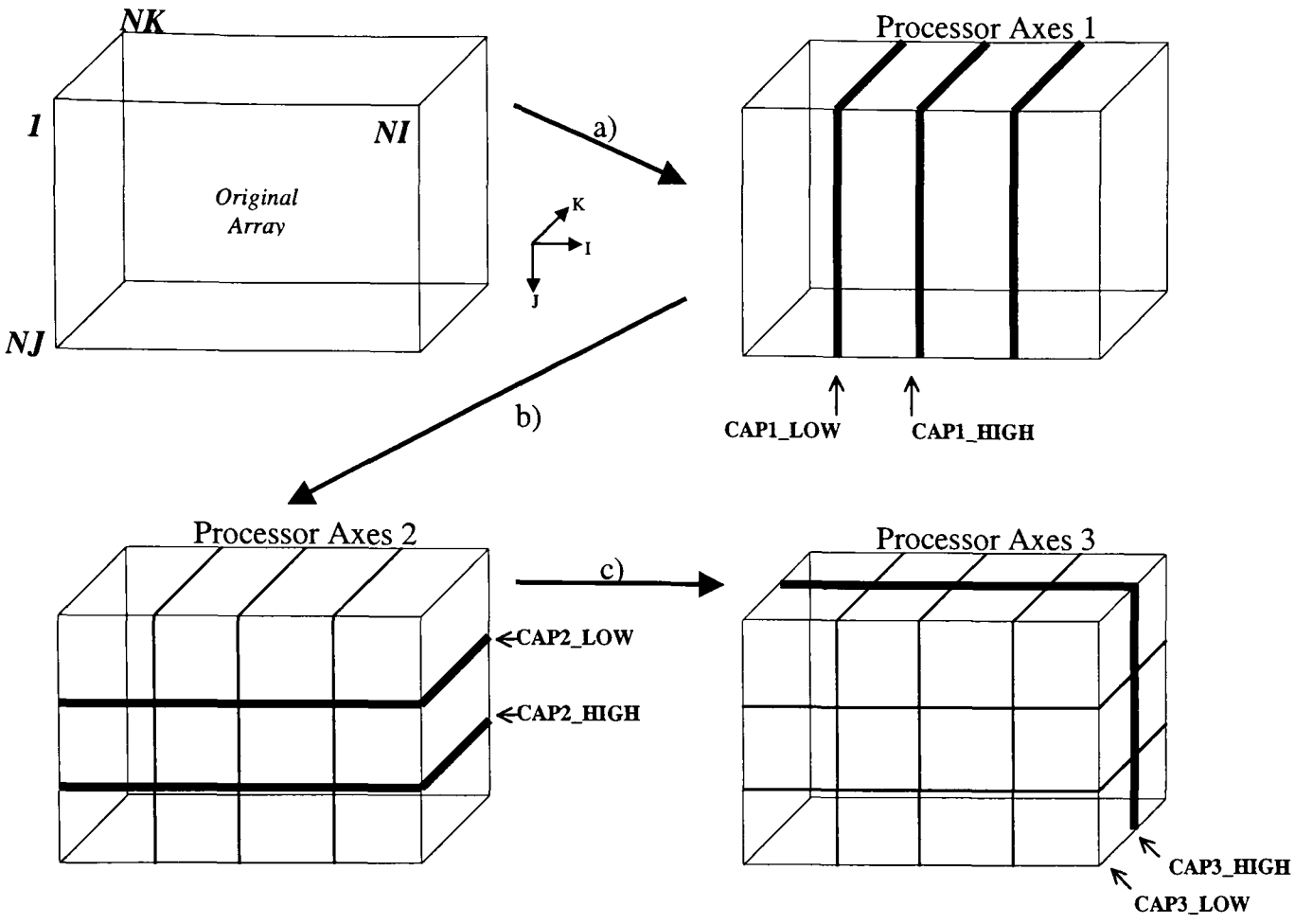


Figure A.3: An example of an array that has a) been partitioned firstly the I direction; b) then partitioned secondly in J direction; and c) finally partitioned in the K direction. The processor axes and partition range limits are shown for each of the different partitions.

Each processor operates on a subset of the domain (its workload), defined by its processor partition range limits CAP_LOW and CAP_HIGH, illustrated in Figure A.3. Each partitioned dimension has its own set of processor partition range limits, which can be uniquely identified by the fact that the processor axes are included within these limits. For example, CAP1_LOW and CAP1_HIGH were generated on the first pass of the parallelisation process using CAPTools, whereas CAP2_LOW and CAP2_HIGH were generated in the second partition, and finally CAP3_LOW and CAP3_HIGH were generated in the third partition. The value of these limits are calculated at runtime using CAP_SETUPPART (for a 1D partition), or CAP_SETUPDPART (for a multi-dimensional partition). The processor partition range limits are evaluated for each partitioned dimension separately, based on the number of processors in that dimension (given by the specified configuration), as demonstrated by Figure A.4, where the processors are grouped as such due to the fact that global limits are used (Section A.2.1). A processor need not have any knowledge of the limits of other processors since the limits are global, meaning neighbouring processors share the same limits in

orthogonal dimensions, and so it only needs to evaluate its own processor partition range limits.

Code uses a 3D partition (where NI=1000, NJ=90, and NK=500):

Main Program
```
CALL CAP_INIT
CALL CAP_SETUPDPART(1,NI,CAP1_LOW,CAP1_HIGH,1)
CALL CAP_SETUPDPART(1,NJ,CAP2_LOW,CAP2_HIGH,2)
CALL CAP_SETUPDPART(1,NK,CAP3_LOW,CAP3_HIGH,3)
```

User specifies a grid4x3x2 configuration (4 columns, 3 rows, and 2 planes, of processors):

	CAP1_LOW	CAP1_HIGH
Processors in column 1	1	250
Processors in column 2	251	500
Processors in column 3	501	750
Processors in column 4	751	1000

	CAP2_LOW	CAP2_HIGH
Processors in row 1	1	30
Processors in row 2	31	60
Processors in row 3	61	90

	CAP3_LOW	CAP3_HIGH
Processors in plane 1	1	250
Processors in plane 2	251	500

Figure A.4: Example demonstrating the initialisation of a parallel code given the specified processor configuration.

A.2.1 Rectangular Partitions

CAPTools uses a simple approach in which each processor owns a rectangular subsection of the domain which are aligned with one another, where 'global' processor partition range limits are used. The limits are said to be 'global' since they are coincidental, which means that each of the processors in a group share the same processor partition range limits in the given dimension. For example, in Figure A.2c Processors 3, 4, and 9, all have the same CAP1_LOW and CAP1_HIGH limits (in the Left/Right direction), and Processors 1, 2, and 3, all

have the same CAP2_LOW and CAP2_HIGH limits (in the Up/Down direction). This approach allows CAPTools to generate neat parallel code without drastically changing the user's code, since simple loop transformations [25] can be used along with other techniques. For example consider the following case in Figure A.5 where the original loop has been partitioned, then the processor partition range limits can easily be incorporated into the loop heading since the partitions are rectangular. The loops still execute from between 1 and NI, and 1 and NJ, where each processor operates between their processor partition range limits (Section B.8 discusses execution control masks). Note that the code is still recognisable even with the processor partition range limits included, which means that the user is still able to maintain and optimise their code.

Original:

```
DO I=1,NI
   DO J=1,NJ
      ...
   END DO
END DO
```

Parallel – rectangular partition used:

```
DO I=MAX(1,CAP1_LOW),MIN(NI,CAP1_HIGH)
   DO J=MAX(1,CAP2_LOW),MIN(NJ,CAP2_HIGH)
      ...
   END DO
END DO
```

Figure A.5: The original loop alongside the parallel loop in which rectangular partitions have been used.

If the limits were not 'global' in every dimension but 'local' (non-coincidental) then non-rectangular partitions would be in use, where the rectangular partitions are not aligned, as seen in Figure A.6. Although this may be better in terms of load balancing, it would be very difficult to calculate these limits generically for a given application code [25]. It would be difficult to calculate the new limits, as CAPTools would have to ensure that there are no 'gaps' in the partition. The main reason why this type of partition is not used is simply because the communication overhead could be very high [23], especially since there would be a lot of overlapping processor partition range limits, where a processor would have to communicate with several processors in a given direction. Another significant reason why this type of partition is not used is because there would be too many changes to the user's code [25], making it difficult to maintain or optimise the code. In the extreme case, shown in Figure A.6, in which the partition is not rectangular, the original loop would have to be

transformed into several loops where the number of loops needed depends on the shape of the partitions used.

Parallel – irregular partition used:

```
DO I=MAX(1,CAP1_LOW1),MIN(NI,CAP1_HIGH1)
   DO J=MAX(1,CAP2_LOW1),MIN(NJ,CAP2_HIGH1)
      ...
   END DO
END DO

...

DO I=MAX(1,CAP1_LOWn),MIN(NI,CAP1_HIGHn)
   DO J=MAX(1,CAP2_LOWn),MIN(NJ,CAP2_HIGHn)
      ...
   END DO
END DO
```

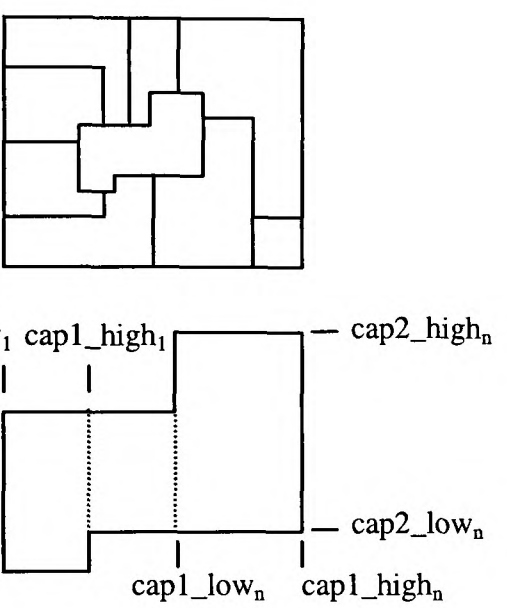


Figure A.6: The parallel loops that are needed instead of the original loop when a non-rectangular partition has been used. Each loop represents a rectangular area within the sub-domain of a processor (which can be seen for the middle processor's first and last rectangular areas).

A.3 Inter-Processor Communication

Although processors mainly operate on data within their own processor partition range limits, it is usually the case that they will often need to use data that is owned by another processor. Consider the example shown in Figure A.7, where a 5-point stencil is being used in a particular calculation. When a 2D partition is used for the same problem, then the domain is dissected into several subsections,

where each processor operates on the data defined by its processor partition range limits. Each processor no longer owns all of the data that they will be using, since when operating on cells on the boundary they will need to use data that is owned by a neighbouring processor. For example, when using the 5-point stencil on the top-right corner cell of Processor 5, it will need to use the values owned by Processors 2 and 4, and similarly values from neighbouring processors will be required for all of the other cells along its boundary.

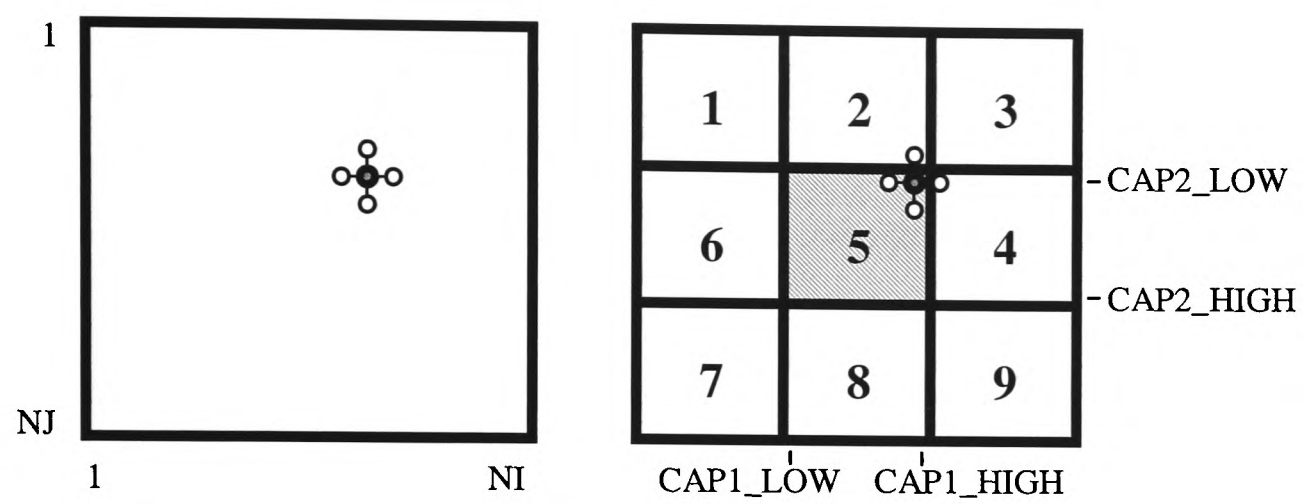


Figure A.7: A 5-point stencil used on the original domain (in serial) and with a 2D partition, where the processor partition range limits have been shown for Processor 5. Neighbouring cells are needed on each processor when applying the stencil to boundary cells.

Communications are therefore necessary when data is required from a neighbouring processor, transferring the requested data onto the processor that needs to use that data, i.e. data is essentially communicated from the processor where it is stored, onto the processor where it is needed. In order to compute in parallel, each processor must have access to the current values of all the data that it needs, which means that a processor will need to receive any data that it does not own itself from another processor before the computation is performed. Note that serial processing has no such overheads associated with it, since communications will never be needed.

As seen in Figure A.8, communications will be needed to update the halo region (data along the boundary) on each processor, where the sample code requiring communication for this and other examples can be seen in Figure A.9. A simple communication structure is needed when using a rectangular partition since each processor will essentially be communicating with a neighbouring

processor whose processor partition range limits coincide. The overlapping area is minimal as each processor only overlaps with a single processor in each direction, making this type of partition efficient, as well as only requiring few changes to the user's code. Note that the parallel code should produce the same results as the serial code (accounting for any round-off that may occur) and so the parallel computations should involve the same data as the serial computations. This means that a processor will need to use the current and up-to-date values of its halo region whenever a computation involves any halo data. This is true for any data that is needed on another processor, as the most current value is required, which means that this data should be communicated before the computation is performed.

Communications are essential in several instances, the most obvious case involving I/O. It has been decided that Processor 1, for reasons of simplicity, should handle all I/O. For example (in Figure A.9), if the dimensions of a mesh had to be read in by every processor, then this could be tedious if hundreds of processors were used, as the user would have to enter the dimensions hundreds of times. This essentially means that once the data is read in by Processor 1, this data will then need to be communicated to the other processors, and similarly, data will need to be passed from each processor to Processor 1 when outputting the data values.

Communications are also essential after processing data whose value needs to be known by all processors. For example, in Figure A.9 every processor evaluates their own copy of the variable SUM, after which a summation is needed, where the total value is calculated and broadcast to all of the processors involved (Section A.3.3.6). In the original code the value of SUM is calculated for I between 1 and NI which is then tested, and so SUM needs to be the summation over the same range when executed in parallel.

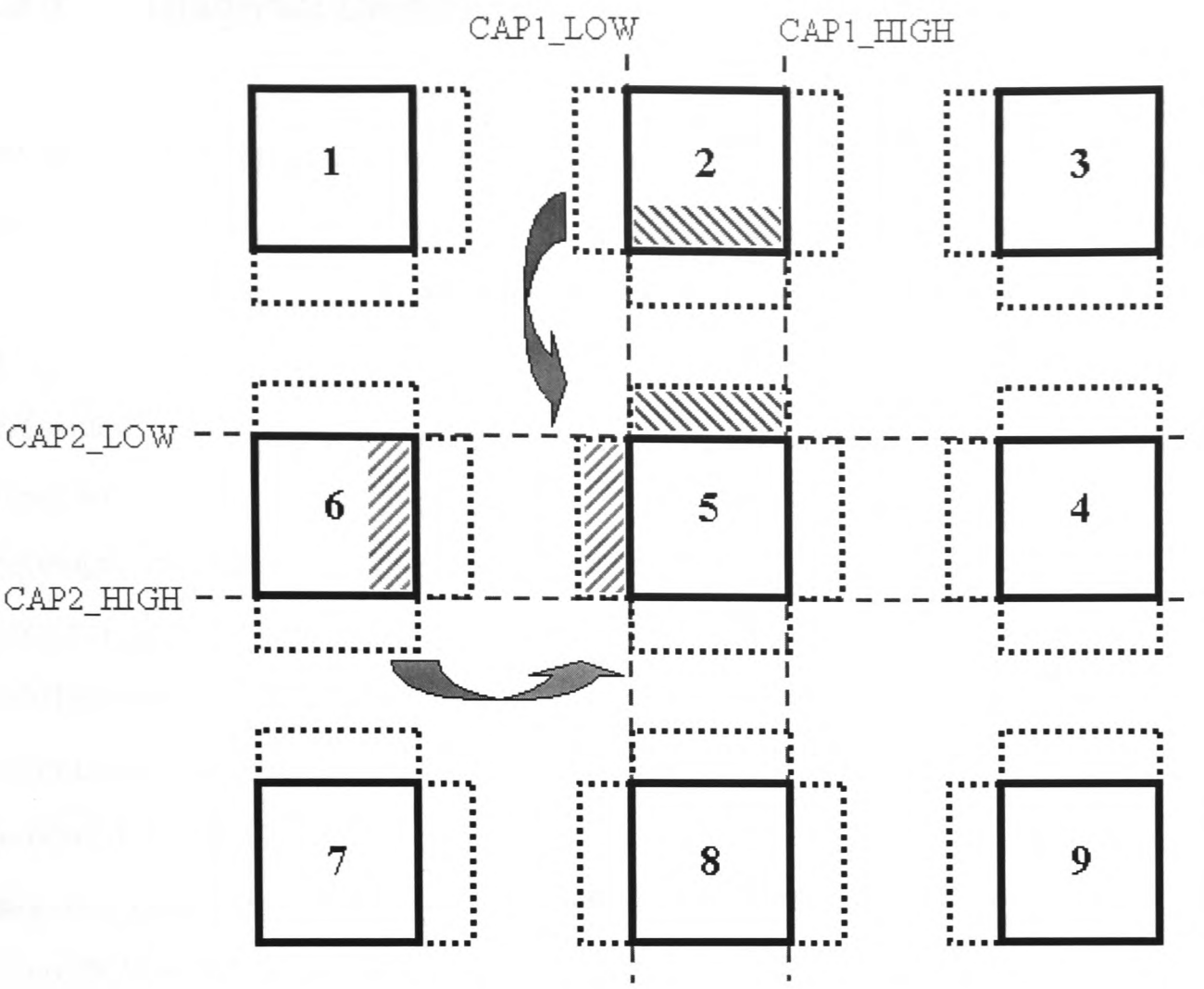


Figure A.8: Updating the processor halo region with values stored on neighbouring processors.

```
DO J=MAX(1,CAP2_LOW),MIN(NJ,CAP2_HIGH)
   DO I= MAX(1,CAP1_LOW),MIN(NI,CAP1_HIGH)
      TNEW(I,J)=( T(I-1,J)+T(I,J+1)+ T(I+1,J)+T(I,J-1) )*.25
   END DO
END DO

IF (CAP_PROCNUM.EQ.1)  READ*,NI,NJ

SUM=0.0
DO I= MAX(1,CAP1_LOW),MIN(NI,CAP1_HIGH)
   SUM=SUM+A(I)
END DO
IF (SUM...) THEN
```

Figure A.9: Sample code in which communications are required. The first example involves using data in the halo region, the second deals with I/O, and the third requires a global summation.

A.3.1 Diagonal Communications

The halo region on each processor will usually have to be updated, meaning that this halo data will need to be obtained from the relevant neighbours. The sequence in which these communications are executed are illustrated in Figure A.10 (a and b), where the first dimension communications in the Left/Right direction are performed before the second dimension communications in the Up/Down direction. If the corner points of the halo region are also required, then this sequence enables the corner points to be included in the Up/Down communications without the need for separate communications [28]. The Left/Right halo region is included in the Up/Down communication, where the communication will start from the beginning of the halo region and have and increased length (Figure A.10 c). Although the corner point is actually owned by a diagonal neighbour, it has already been passed onto the neighbour immediately above/below the requesting processor. This is illustrated more clearly in Figure A.11, where for example, if Processor 5 requests the top-left corner point (owned by Processor 1), then this will first be sent from Processor 1 to Processor 2 during the Left/Right communications. This communicated value will then be sent from Processor 2 to Processor 5 during the Up/Down communications.

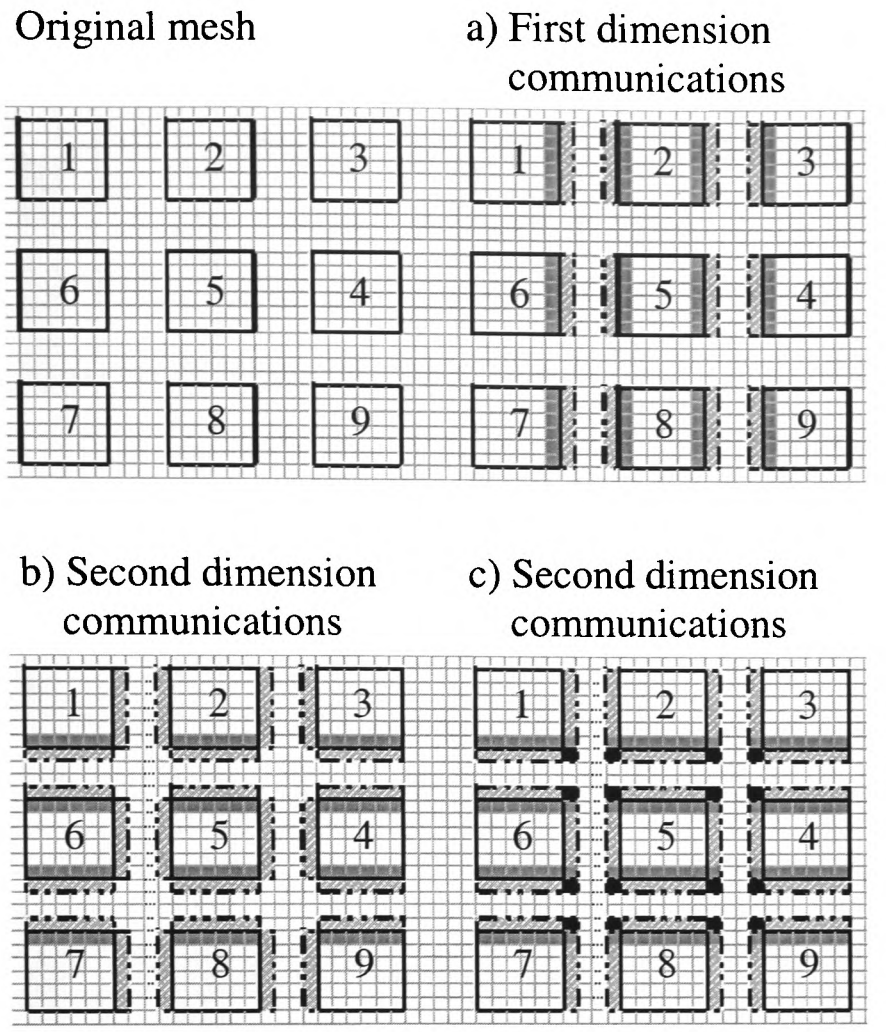


Figure A.10: Sequence of communicating that reduces the number of communications required. Communicate in the direction of those dimensions that were partitioned first, enabling communication of already communicated data.

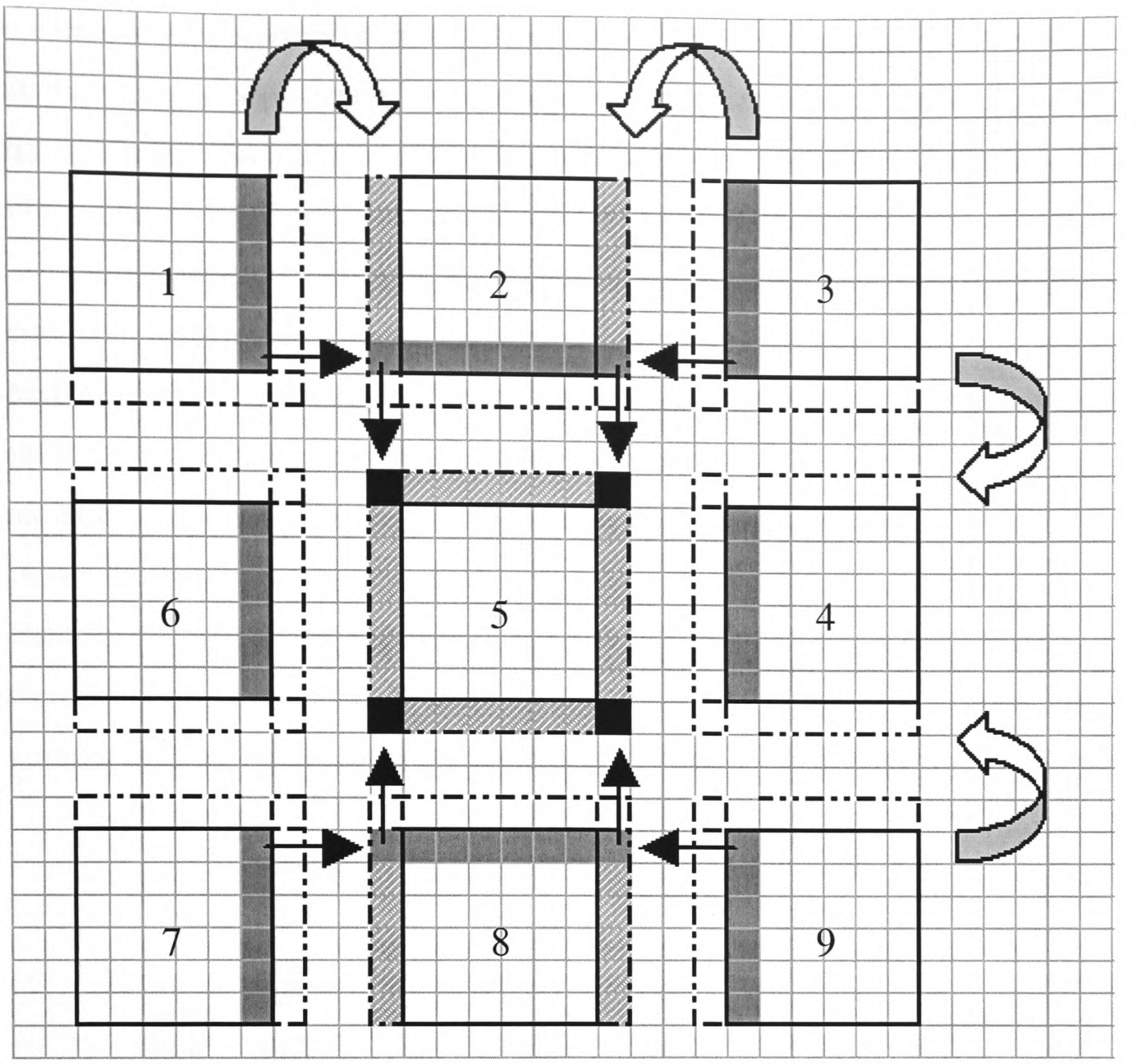


Figure A.11: Example illustrating the communication of corner points when updating the halo region.

A.3.2 Communication Topology

As mentioned earlier, the topology defines the configuration of the processors used, where each processor can be connected to a number of others. The connections between the processors effectively describe the communication topology, where a processor can communicate directly with another processor if they are connected. Ideally the communication overhead should be minimised, which implies that rather than every processor communicating with every other processor, the number of connections needed on each processor should be nominal [26]. Also, to reduce the number of communications it is better to communicate few large messages rather than many small messages otherwise the

communication startup latency can dominate over the computation time [112]. A startup latency is associated with every communication and so it makes sense to try and reduce the number of communications by communicating chunks of data in a single message rather than using several separate messages.

Diagonal communications can be handled using neighbour-to-neighbour communications (Section A.3.1), since the data can be passed vertically and then horizontally, or vice versa. This means that all communications can be handled satisfactorily by only having to communicate with an immediate neighbour, where an example of the communication topology is given for Processor 14 in Figure A.12. Note for example that if Processor 14 wanted to communicate with Processor 2 then a communication would occur between Processors 14 and 5, and then between Processors 5 and 2 (who are neighbours), or alternatively the communication could go via Processor 11.

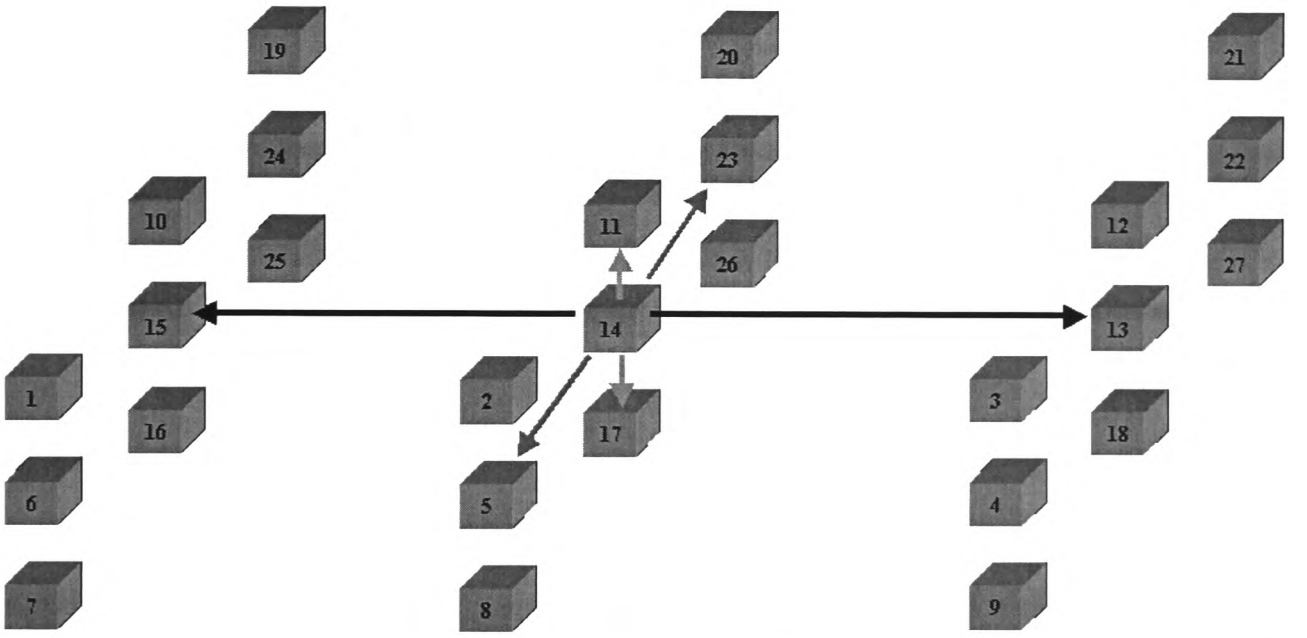


Figure A.12: A 3D mesh example showing the communication topology for Processor 14, which only needs to communicate with its immediately neighbouring processors (15 and 13 in the Left/Right direction, 11 and 17 in the Up/Down direction, and 5 and 23 in the Back/Forth direction).

A.3.3 Generic Communication Utilities

Communications are needed to transfer data from one processor to another, and so several communication utilities have been developed within CAPTools that enable any type of data to be transferred between processors [112, 113], some of which

shall be discussed here. The communications generated by CAPTools are high-level generic communication calls that are mapped onto low-level communication calls using CAPLib (the CAPTools communication library) [112]. They can either be mapped onto communication libraries such as PVM [114], or MPI [115], or onto machine specific communications such as Cray SHMEM [116]. They have been designed to operate on a number of processor topologies (Section A.2), where a minimal number of parameters are used to enable the user to understand the nature of the calls. They are easily portable to other parallel machines [112], and can be effortlessly adapted for use with any other communication library or low-level communication.

A.3.3.1 Send And Receive Communications

The most basic of these communications are CAP_SEND and CAP_RECEIVE, shown in Figure A.13, where the Send statement is used to satisfy the request of the Receive statement. When a processor needs to use data that is owned by another processor (usually a neighbour) it makes a request for that data in the form of a Receive statement, whereby a corresponding Send statement is then needed in order to fulfil the request. Both calls are similar in appearance with just four parameters, differing only in their functionality. The starting address (A) of the communicated data is given, which is used to either indicate from where to start receiving data into, or from where to start sending data out from, depending on the nature of the given communication. The amount of contiguous data in memory being communicated (NITEMS), the data type (ITYPE), and the communication direction (PID) are also given for these two generic communication calls. Note that the PID can also be given as a processor number.

Syntax:

```
CAP_SEND(A,NITEMS,ITYPE,PID)
CAP_RECEIVE(A,NITEMS,ITYPE,PID)
```

Figure A.13: The basic communication calls used by CAPTools to send and receive NITEMS of A which is of data type ITYPE, in the communication direction PID.

These communication utilities can be used to communicate any number of the different data types used within a code, as CAPTools takes advantage of the fact that data is stored as bytes in 1D in memory. The data type (ITYPE) is used internally to convert the data into bytes, which is then used to form an internal communication message that is independent of the data type. Rather than having a different communication call for each of the various data types, it makes sense if a single call could be used, simplifying any maintenance and optimisation that is required. CAPTools represents the different data types using integer values, shown in Table A.1, where for example, when communicating a REAL data variable, then ITYPE will be set to 2, if an INTEGER is communicated then ITYPE is set to 1, and so on.

Data Type:	ITYPE:	CAPTools variable:
Integer	1	CAP_INTEGER
Real	2	CAP_REAL
Double Precision	3	CAP_DOUBLE_PRECISION
Complex	4	CAP_COMPLEX
Logical	5	CAP_LOGICAL
Character	6	CAP_CHARACTER
Byte	7	CAP_BYTE

Table A.1: Data types (ITYPE) in CAPTools.

As demonstrated above, each processor will need to communicate with its immediate neighbour in any given direction, where these neighbours share the same processor partition range limits in the dimensions orthogonal to the communication. It should be noted that the communication topology is essentially the same as the processor topology, in that a communication will only need to occur with a neighbouring processor, which is set up in CAP_INIT. This means that if the processor topology is known, then it is possible to determine who to communicate with simply by using the abstraction of a communication direction. For example, in Figure A.12 Processor 14 will respectively communicate with Processors 15 and 13 when communicating to its Left and Right, with Processors 11 and 17 when communicating Up and Down, and with Processors 5 and 23 when communicating Back and Forth. However, this is true when a grid3x3x3 is used, but may not necessarily be true if a different topology were used. For example, when a grid4x3x2 is used, then Processor 14 will communicate with

Processors 13 and 15 when communicating to its Left and Right, with Processor 19 when communicating Down, and with Processor 2 when communicating Back. In this case there are no neighbours to communication with when communicating in either the Up or Forth direction. The PID is therefore used to represent the direction of communication, as shown in Table A.2. A communication direction, such as CAP_LEFT, CAP_RIGHT, CAP_UP, and CAP_DOWN, can be explicitly used in the communication call, where the processor with whom to communicate with can be determined. For example, when the parallel code is executed on Processor 14, the call to CAP_INIT will set up CAP_LEFT to refer to Processor 15, CAP_RIGHT to refer to Processor 13, and so on, where all of the other processors are setting up their own parameters. This means that when Processor 14 wishes to communicate to its Right, then the communication will automatically be set up to internally communicate with Processor 13.

Direction:	PID:	CAPTools variable:
Left	-1	CAP_LEFT (or CAP_LEFT1)
Right	-2	CAP_RIGHT (or CAP_RIGHT1)
Up	-3	CAP_UP (or CAP_LEFT2)
Down	-4	CAP_DOWN (or CAP_RIGHT2)
Back	-5	CAP_BACK (or CAP_LEFT3)
Forth	-6	CAP_FORTH (or CAP_RIGHT3)

Table A.2: Communication directions (PID) used within CAPTools.

These communications are generic, since they can be used to communicate any data type in any given direction, where the actual communication message is constructed internally. If the PID was not specified, but the actual processor involved was specified, then this implies that the parallel code would only be able to operate using a specific processor topology. Using a generic communication call means that each processor will internally determine whether they need to be involved in the communication, after which the communication is carried out only between those that need to communicate.

For example, in Figure A.14, the value of NI is being received from the Left by all of the processors who need it, which is less complicated than having a Receive statement for each processor. The corresponding Send communication is also given, where one continuous item in memory, of an integer type (ITYPE=1), is being sent to the Right. A single parallel code can be used on any valid

processor topology, eliminating the need for the user to generate a different code for each topology used.

Using a 'neighbour' processor topology rather than a full topology may reduce the overhead in all communications (i.e. little through routing to intermediate processors) and allow communication hardware to operate in parallel. Typically, communications on every processor are performed at the same time. CAPLib maps the logical process topology onto the physical processor topology to take advantage of 'faster' connections. Note that a Full topology is always used in unstructured mesh codes, although communications attempt to be localised [112]. CAPTools communication utilities, such as CAP_SEND for example, check for processor boundaries internally, avoiding conditional statements being generated in the application code and deadlocks. For example, the parallel code will not deadlock when Processor 1 tries to send to, or receive from, its Left.

Example:

```
CALL CAP_RECEIVE(NI,1,1,CAP_LEFT)
CALL CAP_SEND(NI,1,1,CAP_RIGHT)

CALL CAP_RECEIVE(T(CAP1_HIGH+1),1,2,CAP_RIGHT)
CALL CAP_SEND(T(CAP1_LOW),1,2,CAP_LEFT)
```

Figure A.14: Examples of paired communications used to communicate NI and update the upper halo region of the array T.

A.3.3.2 Exchange Communications

When communicating partitioned data, most communications will involve updating the halo region on each processor. Two communication calls could be used to update the halo region on a processor, which are shown in Figure A.14, where the upper halo region (CAP_HIGH+1) of an array T is updated with the lower boundary (CAP_LOW) of its neighbouring processor. The same communication is used to update the halo communication on every processor, where each processor operates using their processor partition range limits.

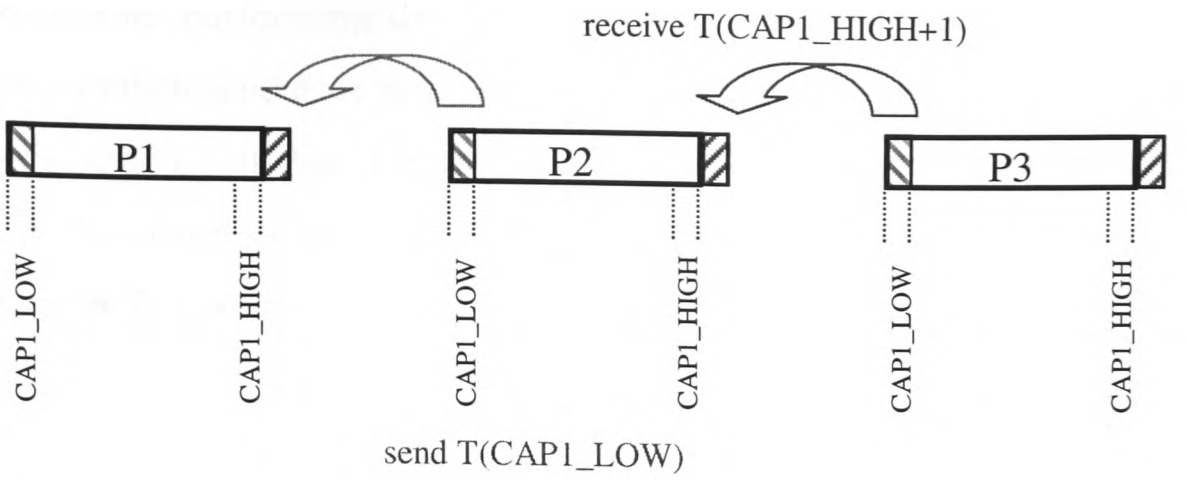


Figure A.15: Update of the upper halo region on every processor by receiving the lower boundary value from the Right neighbour.

In many cases the upper halo regions will have to be updated, as seen in Figure A.15, which involves the use of paired Receive and Send communications. The halo regions may have to be updated on a number of partitioned arrays, which means that the user's code would have several pairs of communications to update each of the halo regions. To perform communications in parallel and simplify the user's code a single Exchange communication call (CAP_EXCHANGE) can be used to perform the same operation as the paired Receive and Send, an example of which is shown in Figure A.16. A similar Exchange would be used to update the lower halo regions in the same way.

Syntax:

```
CAP_EXCHANGE(AIN,AOUT,NITEMS,ITYPE,PID)
```

Example:

```
CAP_EXCHANGE(T(CAP_HIGH+1),T(CAP_LOW),1,2,CAP_RIGHT)
```

Figure A.16: An Exchange communication call, and an example, which is used to exchange data between two neighbouring processors.

The same operation undertaken by the two communications in Figure A.14 can be carried out within the single call shown in Figure A.16, where each processor receives data into their upper halo region from the Right, sending their lower boundary in the opposite direction. The communication parameter list consists of a receiving address (AIN) and sending address (AOUT), along with the other specifications of a normal communication call (NITEMS and ITYPE), where data is received from a neighbouring processor in the specified direction (PID). However, the communication would not operate efficiently if all of the

processors were performing the same action of trying to receive data from one direction and then send data in the opposite direction, as only one processor would initially be sending its data. For example, whilst every other processor has to wait to receive data from its Left, Processor 1 (who has no Left neighbour) can start to send data to Processor 2 on its Right. Processor 2, who has then completed receiving data can then send the necessary data onto its Right neighbour (Processor 3). In this example, the communication time is dependent on the number of processors, and so it was decided that adjacent processors would perform the opposite operation so that the two internal communications worked in conjunction with each other. Therefore a processor will either receive data from its neighbouring processor in the given direction and then send data to the processor in the opposite direction, or vice versa, so that the communications are performed in parallel. For example, if every odd numbered processor were to first receive data from their Left and then send data to their Right, and at the same time every even numbered processor were to first send data to their Right and then receive data from their Left, then the communication time would be independent of the number of processors.

A.3.3.3 Buffered Communications

The examples shown so far have involved communicating data that is contiguous in memory, since the array T has only one dimension. This meant that when updating the halo region, only one cell was communicated, as demonstrated in Figure A.17. When communicating multi-dimensional data these communications cannot be used to communicate non-contiguous data. For example, in Figure A.17, when U is partitioned in the first dimension, then communications will involve non-contiguous data, but when partitioned in the second dimension then communications will involve contiguous data. Buffered communications (CAP_B...) can be used when communicating non-contiguous data. They operate in exactly the same way as the unbuffered communications, where two additional parameters are now needed to indicate the stride (STRIDE) between successive groups of continuous data and the number of strides (NSTRIDE), as seen in

Figure A.18. The term 'paired-index' shall often be used to refer to these two parameters that together describe either an index or group of contiguous indices (or components if 1D-mapped). Note that it is possible to use a buffered communication to emulate the functionality of an unbuffered communication by setting the stride to 1, as demonstrated in Figure A.18. Figure A.19 demonstrates what data would be communicated when using a buffered communication, where the communication message is essentially constructed using the start address, the amount of continuous data, and the number of strides, where the number of bytes is dependent upon the data type.

For example, in Figure A.17 when U is partitioned in the first dimension, Processor 2 needs to receive an entire column of cells into its lower halo region, which is sent from Processor 1. This data is not contiguous in memory and so buffered communications need to be used. The number of continuous cells being communicated is set to 1, where the stride between successive cells is the length of the first dimension (NI), and the number of strides the length of the second dimension (NJ). Note that NITEMS would be set to 2 if two columns of cells needed to be sent. This can be directly compared to the communication used when U is only partitioned in the second dimension, where the number of continuous cells is set to NI. Observe that even though the data is partitioned in the second dimension, the terminology used to describe the direction of the communication is still given as CAP_LEFT and CAP_RIGHT, and not as CAP_UP and CAP_DOWN. CAPTools uses this terminology since this is the first partition, if the picture were rotated then this terminology would still fit.

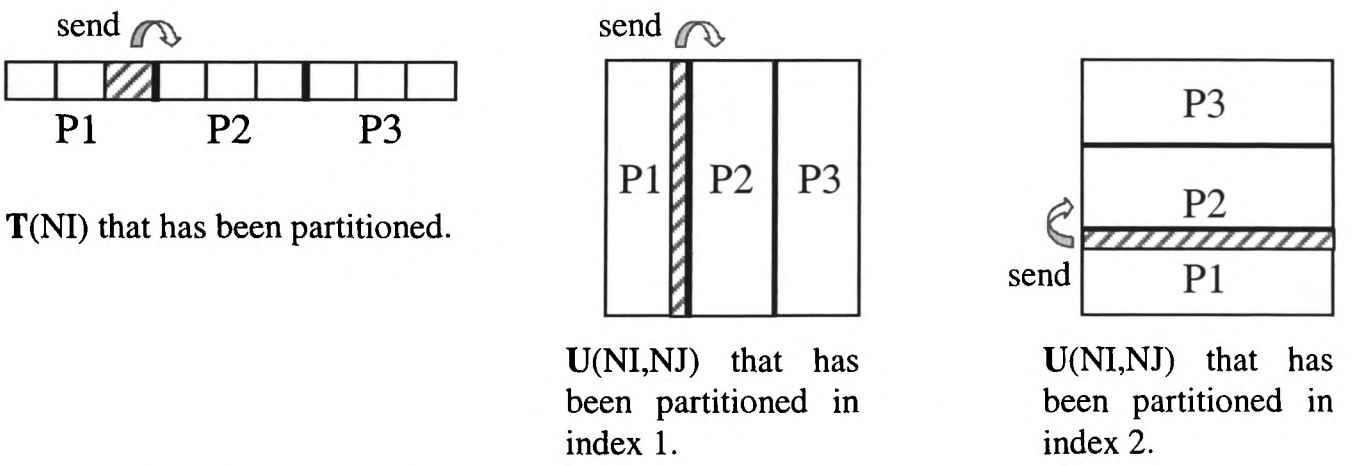


Figure A.17: A 1D array that has been partitioned, along with a 2D array that has been partitioned in index 1, and alternatively in index 2. The lower halo region is updated using the upper boundary of a neighbouring processor. An individual cell is communicated in the first example, a column of cells in the second, and a row of cells (contiguous in memory) in the final example.

Syntax:

CAP_BRECEIVE(A,NITEMS,STRIDE,NSTRIDE,ITYPE,PID) **CAP_BSEND(A,NITEMS,STRIDE,NSTRIDE,ITYPE,PID)**
CAP_BEXCHANGE(AIN,AOUT,NITEMS,STRIDE,NSTRIDE,ITYPE,PID)

Examples:

Communications used when U is just partitioned in index 1:

CAP_BRECEIVE(U(CAP_LOW-1,1),1,NI,NJ,2,CAP_LEFT) CAP_BSEND(U(CAP_HIGH,1),1,NI,NJ,2,CAP_RIGHT)
CAP_BEXCHANGE(U(CAP_LOW-1,1), U(CAP_HIGH,1),1,NI,NJ,2,CAP_LEFT)

Communications used when U is just partitioned in index 2:

CAP_RECEIVE(U(1,CAP_LOW-1),NI,2,CAP_LEFT) CAP_SEND(U(1,CAP_HIGH),NI,2,CAP_RIGHT)
CAP_EXCHANGE(U(1,CAP_LOW-1), U(1,CAP_HIGH),NI,2,CAP_LEFT)

Alternative buffered communications used when U is just partitioned in index 2:

CAP_BRECEIVE(U(1,CAP_LOW-1),1,1,NI,2,CAP_LEFT) CAP_BSEND(U(1,CAP_HIGH),1,1,NI,2,CAP_RIGHT)
CAP_BEXCHANGE(U(1,CAP_LOW-1), U(1,CAP_HIGH),1,1,NI,2,CAP_LEFT)
CAP_BEXCHANGE(U(1,CAP_LOW-1), U(1,CAP_HIGH),NI,1,1,2,CAP_LEFT)

Figure A.18: Buffered communication calls, and some examples relating to the 2D problems shown in Figure A.17.

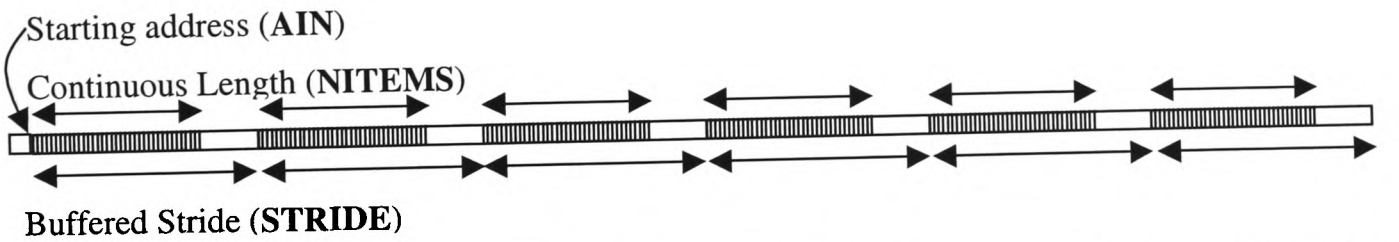


Figure A.19: Representation of communicated data in 1D memory, where a NITEMS of continuous data is communicated NSTRIDE times from the given starting address.

A.3.3.4 Multi-Dimensional Communications

These communication utilities can be used regardless of the number of generated partitions since they are generic, where the same communication call is able to operate correctly on the specified data. The parallelisation procedure is applied to a subsequent dimension when the user decides to partition another dimension using CAPTools (Section B.11). When the user reaches the communication phase of the parallelisation process, not only are new communications calculated and generated, but existing communications are also partitioned. Whilst newly

generated communications (relating to the current partition) will take the previous partitions into account, CAPTools will need to partition any existing communications (from previous dimensions) for the new partitioned dimension.

Consider the example shown in Figure A.17 where just the first index of U has been partitioned. If the user then decides to partition the second index in this example then this means that Up/Down communications may also be needed depending on the requirements of the code. Unlike the communications in Figure A.18 involving an entire row of cells for when U is just partitioned in the second index, these Up/Down communications will typically only involve those cells within the processor partition range limits as seen in Figure A.20 (see Figure A.8 which shows this graphically). Similarly, existing Left/Right communications (generated in the first pass of partitioning) will no longer involve an entire column of cells, and so these communication calls need to be changed. The point to note here is that the previous partitions are taken into account when generating the Up/Down communications, but the Left/Right communications are actually changed to account for the new partitioning, as they have already been generated (i.e. they are not being created afresh).

```
CAP_BEXCHANGE(U(CAP1_LOW-1,CAP2_LOW),
              U(CAP1_HIGH,CAP2_LOW),1,NI,
              CAP2_HIGH-CAP2_LOW+1,2,CAP_LEFT)

CAP_EXCHANGE(U(CAP1_LOW,CAP2_HIGH+1),
             U(CAP1_LOW,CAP2_LOW),
             CAP1_HIGH-CAP1_LOW+1,2,CAP_DOWN)
```

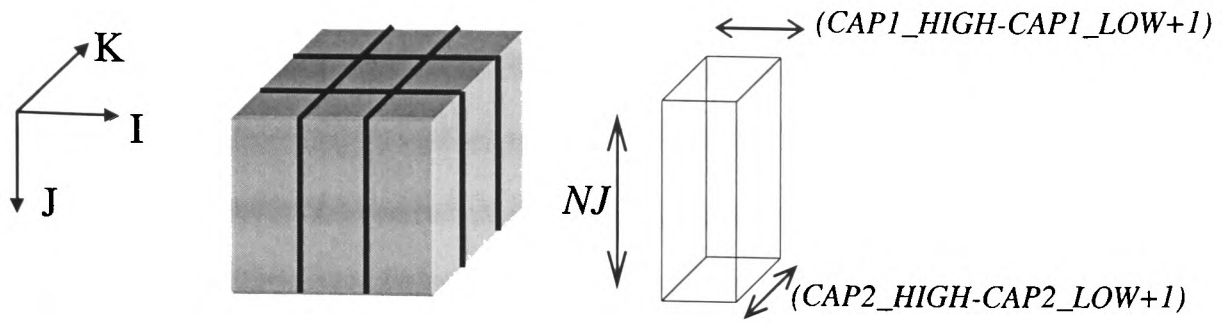
Figure A.20: The Exchange communications that are used to update the halo regions on each processor in Figure A.8, where the width of the halo region is 1.

The communications shown in Figure A.20 which are used to update the halo region when partitioned in 2D can be compared directly with those communications shown in Figure A.18 in which a 1D partition was used on the first index. The starting address to receive into, and send from, for the buffered Exchange is the same in the first dimension (CAP1_LOW-1, and CAP1_HIGH, respectively). However, the starting address in the second dimension is now partitioned, which means that the communication no longer starts from the lower declared limit of the second dimension (1 in this case), but it starts from the lower processor partition range limit (CAP2_LOW). The number of continuous items

(NITEMS) is set to 1, where the stride to the next set of continuous items (STRIDE) is set to NI. The other difference between the altered communications in Figure A.20 and those shown in Figure A.18 is notably the number of strides (NSTRIDE). The entire column of cells (NJ) that was being communicated previously due to a 1D partition is now itself partitioned, which means that each processor will only communicate data between its processor partition range limits in the second dimension (CAP2_HIGH-CAP2_LOW+1).

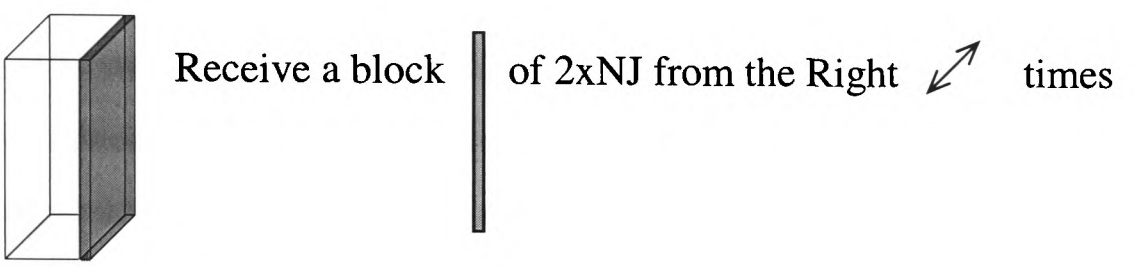
When a 1D partition was used on the second dimension of U in Figure A.17, the lower halo region was updated by communicating contiguous data in the direction of the lower halo region. Although the upper halo region is updated in Figure A.20 in the second dimension, the communication call used is very similar in appearance. The starting address now starts from the lower processor partition range limit in the first dimension rather than 1, and the number of continuous items (NITEMS) is the length of the processor partition range limits in the first partitioned dimension (CAP1_HIGH-CAP1_LOW+1). The amount of data being communicated is essentially governed by the partition length. Note that had the halo width been set to 2 say, then a buffered Exchange could have to be used in the second communication in Figure A.20, where the number of continuous items is the same, the stride is NI, and the number of strides is 2.

Currently the buffered communications can be used when communicating non-contiguous data in one of the partitioned dimensions, however, when the data is not contiguous in another dimension then CAPTools will handle this by placing a buffered communication inside a DO Loop. Most codes use a 2D partition, and so the data will usually only have to be buffered in two dimensions. For example, consider the case given in Figure A.21 where a whole plane of data is being received from the Right for a 3D variable T(NI,NJ,NK) that has first been partitioned in the I dimension, and then the K dimension. Again for demonstration purposes, let the halo width be set to 2, such that a plane of 2xNJ is being received from the Right, CAP2_HIGH-CAP2_LOW+1 times (Figure A.21a). Similarly, when receiving data in the orthogonal direction (in the Down direction), 2 planes of NJx(CAP1_HIGH-CAP1_LOW+1) is received (Figure A.21b).



a) USAGE: **=T(I+1,J,K)+T(I+2,J,K)**

```
DO K=CAP2_LOW,CAP2_HIGH
   CAP_BRECEIVE(T(CAP1_HIGH+1,1,K),2,NI,NJ,2,CAP_RIGHT)
END DO
```



b) USAGE: **=T(I,J,K+1)+T(I,J,K+2)**

```
DO K=(CAP2_HIGH+1),(CAP2_HIGH+2)
   CAP_BRECEIVE(T(CAP1_LOW,1,K),(CAP1_HIGH-CAP1_LOW+1),
                NI,NJ,2,CAP_DOWN)
END DO
```

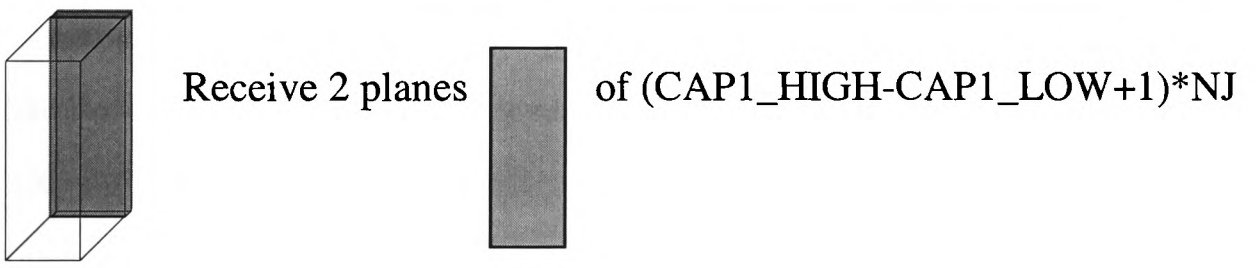


Figure A.21: Example showing when it is necessary to communicate a single plane at a time. The communicated data is not contiguous in more than one dimension.

A.3.3.5 Broadcast Communications

Unlike the above communication types where data is just communicated between neighbouring processors, Broadcast statements can be used when data needs to be known by all processors. There are two types of broadcast statements, CAP_MBROADCAST and CAP_BROADCAST (Figure A.22), each

broadcasting NITEMS, of type ITYPE, of A to the processors involved. The first call is used to broadcast the value of A from the master to all other processors, whereas the second call is used to broadcast the value of A to processors from the processor who calls this routine with IOWNER set to True.

Syntax:

```
CAP_MBROADCAST(A,NITEMS,ITYPE)
CAP_BROADCAST(A,NITEMS,ITYPE,IOWNER)
```

Figure A.22: BROADCAST utilities, and an example in which partitioned data is assigned in several instances.

Broadcasts are needed in several instances, the first being for unmasked usage (see Section B.8), where the data has been assigned on a particular processor but is then used by all processors, in which case the owning processor needs to broadcast this data to the other processors. The second case is when dealing with I/O so that only one processor handles this, and the third being when there is a conflict assignment (Section B.9.1.3), where it is not known for certain who owns the assigned data.

It is important to note that CAPTools does not actually generate Broadcast statements in a structured mesh code but instead generates a combination of Send and Receive statements as most broadcasts only need to involve immediate neighbours (which is more efficient). In Figure A.23 for example, the assignment of V(X) is made on the processor owning the value of X and is then broadcast to neighbouring processors using Send and Receive communications.

```
Assignment of V(X) made on processor where CAP1_LOW<=X<=CAP1_HIGH

Broadcast data to the Left
IF (X.GE.CAP1_LOW .AND. X.LE.CAP1_HIGH) THEN
   CAP_SEND(V(X),1,2,CAP_LEFT)
END IF
IF (X.GT.CAP1_HIGH) THEN
   CAP_RECEIVE(V(X),1,2,CAP_RIGHT)
END IF
IF (X.GT.CAP1_HIGH) THEN
   CAP_SEND(V(X),1,2,CAP_LEFT)
END IF

Broadcast data to the Right
IF (X.GE.CAP1_LOW .AND. X.LE.CAP1_HIGH) THEN
   CAP_SEND(V(X),1,2,CAP_RIGHT)
END IF
IF (X.LT.CAP1_LOW) THEN
   CAP_RECEIVE(V(X),1,2,CAP_LEFT)
END IF
IF (X.LT.CAP1_LOW) THEN
   CAP_SEND(V(X),1,2,CAP_RIGHT)
END IF
```

Figure A.23: Example illustrating how a combination of Send/Receive communications can be used to broadcast data to neighbouring processors.

A.3.3.6 Commutative Communications

In parallel each processor operates upon their workload, but it is often necessary to find the maximum or summation of an array for example, as demonstrated in Figure A.24. Rather than communicate the entire array to every processor who then sums the array, it makes sense to summate the localised workload on each processor separately and then add these subtotals. This can be achieved using a Commutative communication call (CAP_COMMUTATIVE), where a specified operation (FUNC) is performed on the given data (VALUE), of type ITYPE. For example, consider the case where a row of processors need to summate T, then the local value of SUM is calculated on each processor, after which a commutative is used to obtain the global value of SUM, which is the summation of all the processors. This Commutative call involves all of the processors in the topology, but when partitioned in more than one dimension (using a Grid or Torus) the communication direction may be used to apply the Commutative across a processor dimension. For example, the sum for all of the processors in a column

of processors may be needed, and so the CAP_UP direction can be specified as the IDIR parameter when using CAP_DCOMMUTATIVE.

```
CAP_COMMUTATIVE(VALUE,ITYPE,FUNC)
CAP_DCOMMUTATIVE(VALUE,ITYPE,FUNC,IDIR)
```

Serial:

```
SUM=0.0
DO I=1,NI
   SUM=SUM+T(I)
END DO
```

Parallel:

```
SUM=0.0
DO I=MAX(1,CAP1_LOW),MIN(NI,CAP1_HIGH)
   SUM=SUM+T(I)
END DO
CALL CAP_COMMUTATIVE(SUM,2,RADD)
```

Figure A.24: Example in SUM is the summation of the array T, which is partitioned. After each processor calculates their local value of SUM, the commutative adds these together and broadcasts the value to all processors involved, such that each processor has the global value of SUM after the commutative.

A.4 Compiling And Executing CAPTools Generated Parallel Code

A single script command (capmake) can be used to compile the generated parallel code as seen in Figure A.25, where several options such as '-m' to specify the machine type and '–p' to specify the parallel environment may be used if the default options are not desired (see the CAPTools User Manual [113]). CAPTools is capable of operating on environments such as PVM3, MPI, MPICH, and Cray SHMEM, on machines such as the SUN, SP2, T3D, T3E, ORIGIN, NEC, FUJITSU, and on the DEC Alpha.

The processor topology (Figure A.2) to be executed on is specified at runtime using the caprun script with the '-top' option. A 4x2 grid topology has been chosen in the example shown in Figure A.25, where the user has decided that there will be 4 processors in the first partitioned dimension (CAP_DNPROC(1)=4), and that there will be 2 processors in the second partitioned dimension (CAP_DNPROC(2)=2).

capmake [options] <sourcefiles> <objectfiles> or codename

e.g.: capmake –m sun –p pvm3 fabpar.f fabpar_sun
Compiles fabpar.f to fabpar_sun linking CAPLib Sun PVM 3.x libraries

caprun [options] codename

e.g.: caprun –m sun –p pvm3 –top grid4x2 fabpar_sun
Runs fabpar_sun under PVM 3.x using a 4 by 2 processor topology

Figure A.25: Scripts used to compile and execute a CAPTools generated parallel code.

A.5 Summary

The aim of this Appendix was to provide background knowledge of the CAPTools SPMD parallelisation strategy and the CAPLib communication library. The reasons behind the development of CAPTools were discussed, revealing its ethos of providing a useful service to the user. The processor configuration, communication topology, and generic communication utilities were discussed along with the reason why rectangular partitions were used. The issues discussed in this Appendix, and the criteria of CAPTools that were stated in Figure 1.3, act as a foundation for Chapters 2 through to 7, particularly putting Chapter 2 (relating to the DLB strategy) into context.

Appendix B CAPTools Algorithms And Data Structures

Having discussed the fundamental components of a CAPTools parallelisation in the previous Appendix, such as the manner in which data is partitioned, and the generic communication library utilities, this Appendix attempts to provide an in-depth investigation of CAPTools. The various stages involved in producing an efficient parallel code from a serial Fortran 77 code are outlined along with algorithms explaining how each stage is accomplished. Knowledge of the algorithms and data structures used by CAPTools will prove useful in the understanding of the DLB strategy that shall be automated within CAPTools.

B.1 The Parallelisation Of A Structured Mesh Code Using CAPTools

The stages involved in generating a parallel code using CAPTools follow, where the automatic parallelisation of a code follows the equivalent manual practice of parallelising a code:

1) *The user loads the serial Fortran 77 code into CAPTools*
2) *A detailed dependence analysis is performed on the serial code*
3) *A data partition for one array is prescribed by the user and inherited throughout the code to a comprehensive set of arrays*
4) *Execution control masks are generated*
5) *Communication requests are made, migrated up the code and merged together, after which the communication calls are generated*
6) *The final code is generated*

The main stages 3 to 5 inclusive are iterative, meaning that they are repeated each time that the user partitions another dimension, as demonstrated in Figure B.1. After each main stage it is suggested that the user saves a database of their current parallelisation stage, from which it is possible to continue the parallelisation stage

without having to start from the beginning again. For example, the user can save the database after the dependence analysis stage and then proceed to select a particular index of a certain variable for partitioning. The user may then feel that it would be better to partition a different index instead, and so rather than perform the dependence analysis again they can simply load in the saved database and proceed from there.

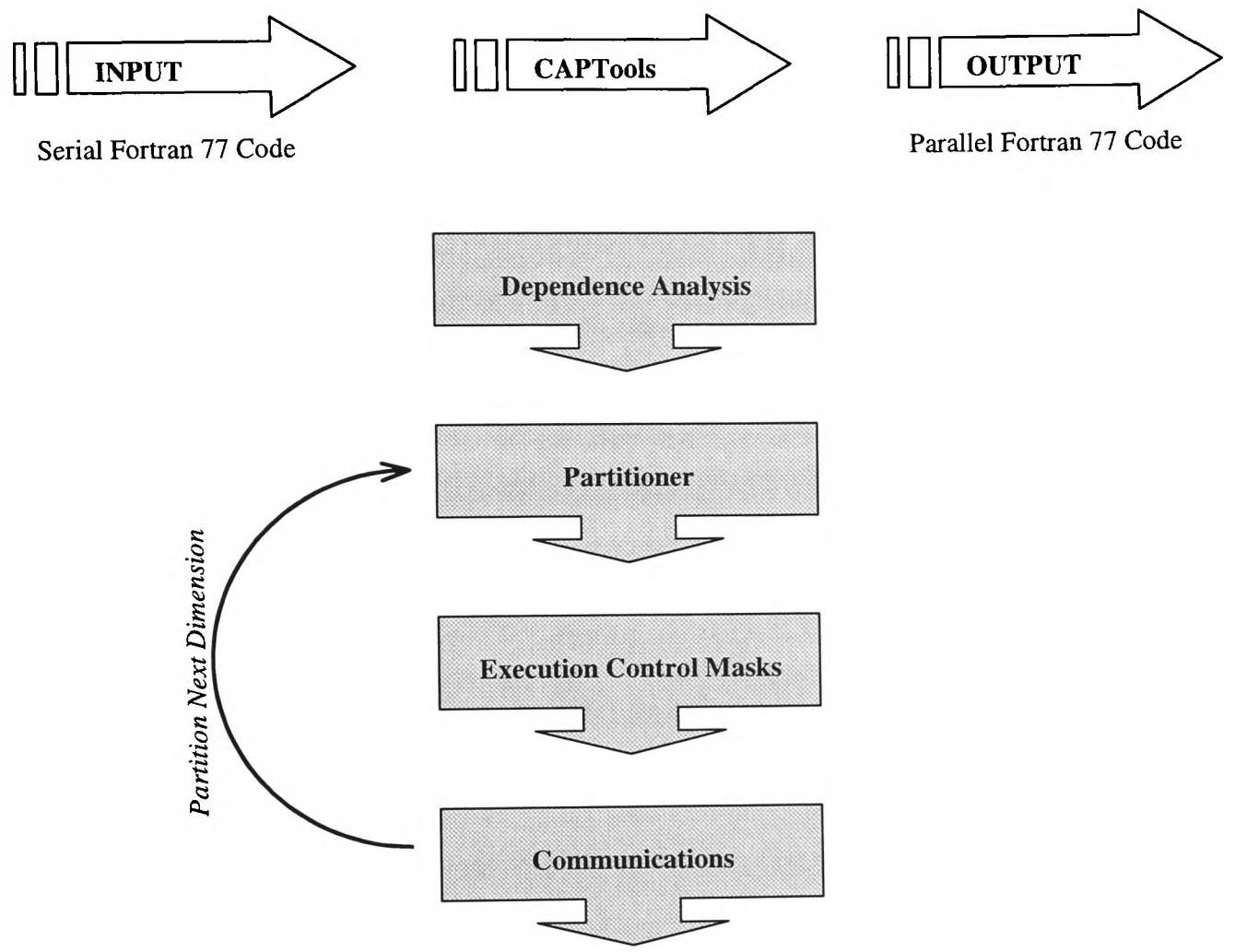


Figure B.1: Representation of the parallelisation stages used within CAPTools.

B.2 Loading The Serial Code

The serial Fortran 77 code can be read into CAPTools using the 'Load Fortran 77 Source ...' from the File option in the main CAPTools window (Figure A.1). The serial source code is parsed [117, 118], from which symbol tables, parse trees, routine call graphs, and control flow graphs, can be constructed. Each routine has its own symbol table containing all of the source variables and also the source symbols (such as IF's, DO's, END's, and CALL's for instance). The parse tree is

constructed as a binary tree consisting of nodes that refer to a symbol table entry with left and right branches pointing to the next nodes. Each SYMBOL data structure has several fields associated with it. A NAME field is used to store the symbol name, a KIND field is used to indicate the kind of data that the symbol is, and a HASH function value for this symbol (the row of the hashed symbol table in which this symbol is stored) is used to enable quick access to this symbol in memory. For example, Figure B.2 shows a simple parse tree from CAPTools for the integer assignment statement T=T-1 in which the = symbol has a KIND=KEYEQUALS, the T symbol has a KIND=KEYINTEGER, and the 1 symbol has a KIND=KEYCONSTANT. The KIND field is important because it allows CAPTools to identify key symbols, such as IF statements for example (KIND=KEYIF), or call statements (KIND=KEYCALL).

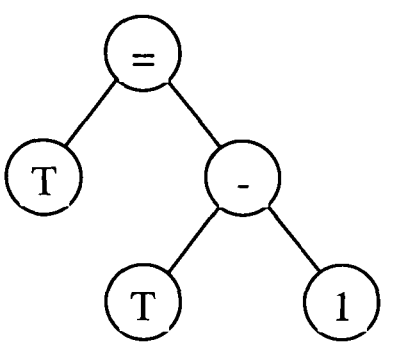


Figure B.2: A parse tree within CAPTools that represents an assignment statement (involving integers).

B.3 The Call Graph

A routine call graph can be used to represent the structure of the code, where each node represents a particular routine in the code, and each edge represents the connection between routines. For example, if routine Sub1 in Figure B.3 calls routines Sub2, and Sub3, then there will be a connection from node Sub1 to node Sub2 and Sub3. The routine call graph can be constructed by identifying calls to routines in the parse tree and then matching them with the relevant routine header. Each routine has a link to the next routine in the order as they were read from the input file, where TOPOFFILE is the first routine in the file, and subsequent routines can be listed using NEXT. For example, in Figure B.4 the TOPOFFILE would be Main where the next routine would be Sub1, then Sub3, and then Sub2.

Performing a depth first search for every routine call, starting from the main program, can yield the strict order of the call graph, where the routines are stored in reverse order. Only after every routine that is called from within a routine has been added to the list, can that particular routine also be added to the ordered list. For example, Sub1 in Figure B.3 can only be added to the list after both Sub2 and Sub3 have been added. This strict order call graph can then be used for interprocedural analysis of the dependence graph (see Section B.6), where all of the called routines within a particular routine will have already been processed, since the called routines are listed before the calling routine. In Figure B.4 for example, the routines can be processed in their strict order by setting TOPROUTINE to Sub3, and using STRICT to process all of the routines above this in the call graph (Sub3→Sub2→Sub1→Main). Using the strict call graph will also be relevant when determining where to place the DLB code (see Section 5.7), as the strict ordering of the routines is employed when traversing through the routine boundaries during interprocedural traversal.

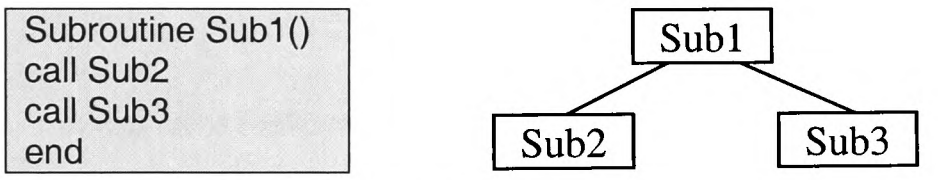


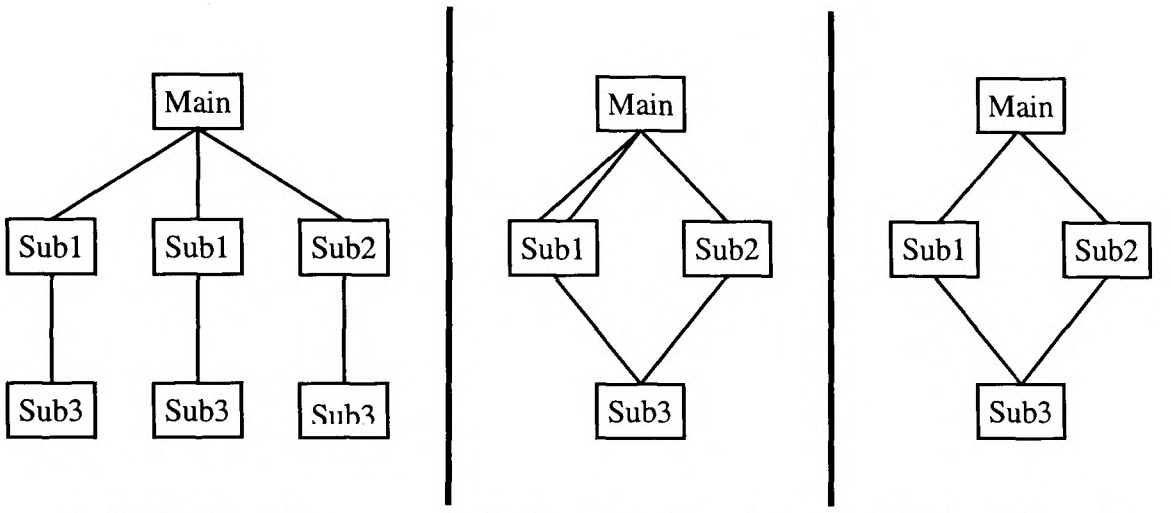
Figure B.3: Sample code with the associated call graph.

A routine may be called several times throughout the code, possibly numerous times from within the same routine, where each instance of the call must be treated differently, as demonstrated in Figure B.4. In this example there are two calls to Sub1 from within Main, where the first call is made using t and the second call is made using n. There are also two separate calls to Sub3 from inside Sub1 and Sub2, Following on from the example given in Figure B.3, there will be three individual branches from Main to each of its called routines (Sub1, Sub1, and Sub2), and another branch from these routines to Sub3.

Visually representing the structure of the code in this manner within CAPTools would become complicated if a larger code were given, where there would be many branches for each individual 'route' in the call graph. The call graph can therefore be simplified by only including each routine once, where the different connections are shown, but this too can be simplified further. The call

graph for this sample code is therefore represented by the last call graph shown, where each routine is included only once, and the connection between routines is only shown just once. Note that CAPTools actually stores the complete call graph internally and that the simplified graph is only used for visualisation. An example of how STRICT and NEXT can be used to process the call graph is also shown in Figure B.4. An example call graph displaying 26 routines within CAPTools is shown in Figure B.5, which the user is able to examine.

```
S0    Main
S1    call Sub1(t)
S2    call Sub1(n)
S3    call Sub2(n)

S4    Subroutine Sub1(x)
S5    call Sub3(x)

S6    Subroutine Sub3(r)
S7    read*,r

S8    Subroutine Sub2(c)
S9    ...=c
S10   call Sub3(v)
```




```
TOPOFFILE                                   (Listed in the order read in from file)
Main  →   NEXT
          Sub1  →   NEXT
                    Sub3  →   NEXT
                              Sub2

TOPROUTINE                                  (Listed in strict order)
Sub3  →   STRICT
          Sub2  →   STRICT
                    Sub1  →   STRICT
                              Main
```

Figure B.4: Example demonstrating that the value of t does not always equal the value of n. The sample code and its call graph are shown (in various degrees of simplicity). A demonstration of how the routines in this example would be processed is also shown.

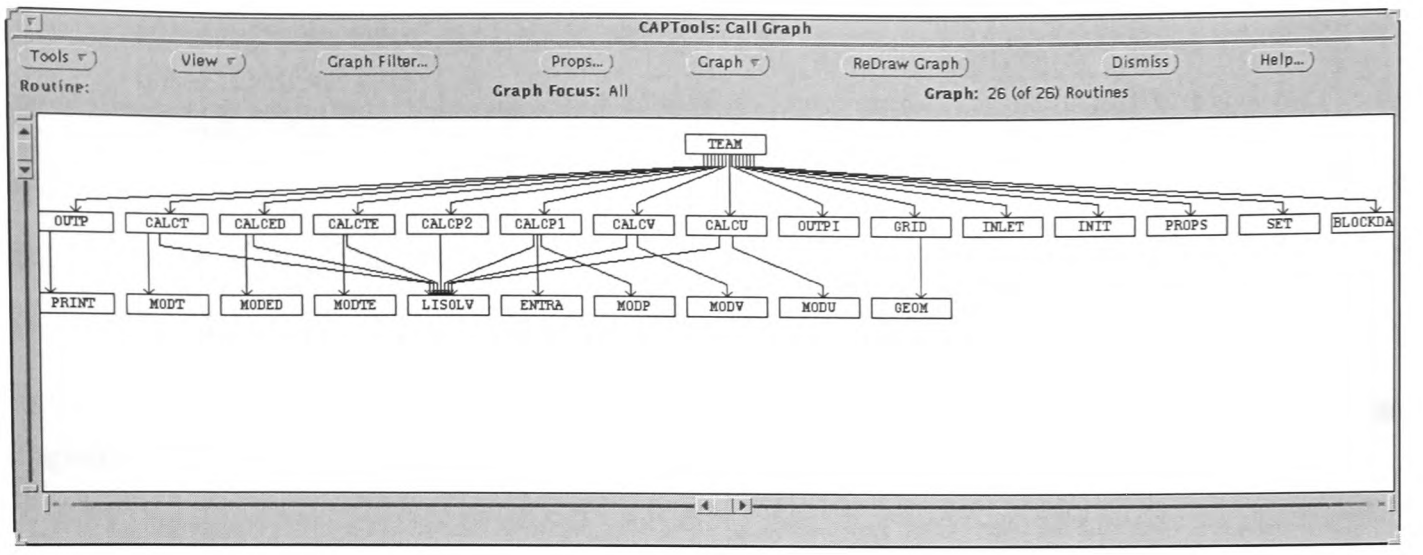
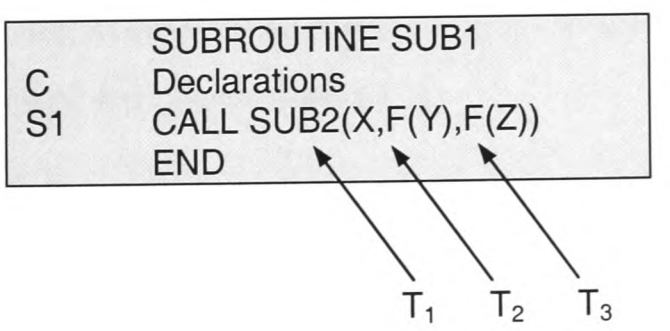


Figure B.5: Example of the Call Graph window in CAPTools showing 26 routines.

Each routine node (ROUTINE) stores a list of routines that this particular routine calls (CALLS), and a list of routines that have called this routine (CALLEDBY). A routine may call several other routines, where some routines may be called more than once. Similarly, a routine may be called by several other routines, where some routines may call this routine more than once. Considering the called routines of a particular routine, each instance (a call to a routine or function) can be stored within the CALLS data structure, where REF stores the called routine, COMMAND stores the statement containing the call, and TREE contains the parse tree of the routine reference in the caller. An example demonstrating the CALLS data structure can be seen in Figure B.6 for the routine SUB1, where a call is made to SUB2 which includes F(X) and F(Y) as some of the parameters. The pseudo code in Figure B.7 shows how CAPTools uses this data structure to traverse interprocedurally through the call graph, which in this case are the called routines.



REF=SUB2	REF=F	REF=F
COMMAND=S1	COMMAND=S1	COMMAND=S1
TREE=T_1	TREE=T_2	TREE=T_3

Figure B.6: Example illustrating the CALLS data structure for the routine SUB1, in which a call to SUB2 is made (whose parameters include calls to the function F).

```
PROCEDURE DFSCALL(ROUTINE);
CALLS:=ROUTINE^.CALLS;
WHILE (CALLS <> NIL) DO
  BEGIN
  ...
  DFSCALL(CALLS^.REF);
  ...
  CALLS:=CALLS^.NEXT;
  END;
```

Figure B.7: Pseudo code used to interprocedurally traverse the call graph.

B.4 The Control Flow Graph

The control flow graph (CFG) stores the code structure for each routine, where the CFG consists of a set of nodes that represent a block (or group) of statements, known as a basic block, with directed control flow paths from one node to another [118]. Control flows directly through all of the statements in each block, after which control passes to the next block of statements. For example, if the flow of control is always going to flow to statement 2 from statement 1 then both of these statements can be grouped together into a block. Blocks are used rather than individual statements for the reason that it simplifies the CFG. For example, the statements of the given code can be divided into blocks as demonstrated in Figure B.8, where the associated control flow graph can be seen in Figure B.9. Control from Block 1 may flow either to Block 2 or to Block 5, i.e. either to another iteration of the I loop, or to the statement after the loop when there are no further iterations. The flow of control in Block 3 loops back (a backlink) to Block 2, re-iterating the J loop. If the condition in Block 5 is True then control will flow to Block 6, otherwise control will flow to Block 7.

No.		Block	Loop Nesting
S1	DO I=2,NI	Block 1	Loop S1
S2	DO J=1,NJ	Block 2	Loop S1,S2
S3	A(I,J)=	Block 3	Loop S1,S2
S4	B(I,J)=	Block 3	
S5	END DO	Block 3	
S6	C(I)=	Block 4	Loop S1
S7	END DO	Block 4	
S8	IF (CONDITION1) THEN	Block 5	
S9	C(I)=	Block 6	
S10	ELSE	Block 7	
S11	C(I)=	Block 7	
S12	END IF	Block 7	
S13	IF (CONDITION2) THEN	Block 8	
S14	GOTO 10	Block 9	
S15	END IF	Block 9	
S16	DO J=1,NJ-1	Block 10	Loop S16
S17	A(1,J)=	Block 11	Loop S16
S18	B(1,J)=	Block 11	
S19	END DO	Block 11	
S20	10 CONTINUE	Block 12	
S21	A(1,NJ)=	Block 12	
S22	B(1,NJ)=	Block 12	

Figure B.8: Code to demonstrate control flow.

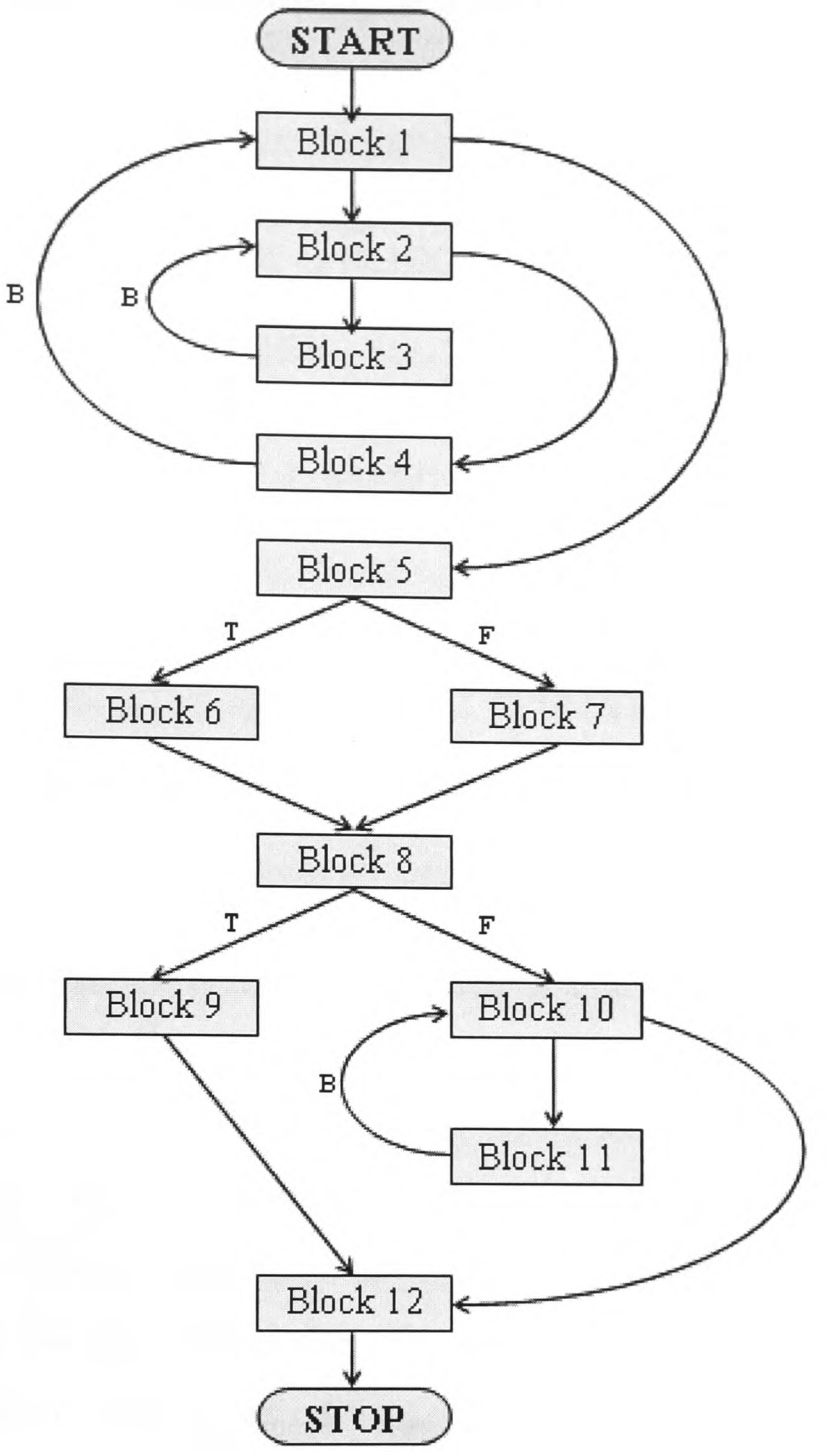


Figure B.9: Control Flow Graph for example given in Figure B.8 above (T=True, F=False, and B=Backlink).

The first block of a routine is stored in BLOCKTOP within the ROUTINE data structure, which is connected to the first block in that routine. These blocks of statements are stored within CAPTools as a BLOCK data structure, where each one of these blocks will point to a list of the statements (COMMAND) that belong to this BLOCK. The pseudo code shown in Figure B.10 illustrates how it is possible to traverse through the entire code in the order of the original files. Starting with the first statement in the block, each statement within that block is processed (where COMMAND^.NEXT links to the next statement in the given block), after which the next block of statements can be processed (using BLOCK^.NEXT which is the order as read from file). Once all of the blocks

within a routine have been processed, then the next routine can be processed in a similar manner.

```
CURRENT_ROUTINE:=TOPOFFILE;
WHILE (CURRENT_ROUTINE <> NIL) DO
  BEGIN
  CURRENT_BLOCK:=CURRENT_ROUTINE^.BLOCKTOP;
  WHILE (CURRENT_BLOCK <> NIL) DO
    BEGIN
    CURRENT_COMMAND:=CURRENT_BLOCK^.COMMAND;
    WHILE (CURRENT_COMMAND <> NIL) DO
      BEGIN
        ...
        ...
      CURRENT_COMMAND:=CURRENT_COMMAND^.NEXT;
      END;
    CURRENT_BLOCK:=CURRENT_BLOCK^.NEXT;
    END;
  CURRENT_ROUTINE:=CURRENT_ROUTINE^.NEXT;
  END;
```

Figure B.10: Pseudo code used to traverse every statement in the input code.

Each BLOCK contains information regarding the HASFATHER and the HASCHILD data structures, which respectively represent a list of blocks through which the flow of control must have passed in order to reach this particular block, and from which the flow of control will pass to from this particular block. For example, in Figure B.9 Block 8 will have Block 6 and Block 7 in its HASFATHER list, and Block 9 and Block 10 in its HASCHILD list. The control flow graph is doubly-linked, where Block 8 is a child of Block 6, and Block 6 is a father of Block 8.

In order to look at possible statements from which the control could have flowed down through, it is useful to consider the HASFATHER data structure. A depth first search (DFS) can be performed from a starting block (STARTBLOCK), passing through all blocks marking all reachable blocks up the control flow graph, and similarly it is possible to perform a DFS down the control flow graph using the HASCHILD data structure of the block. Figure B.11 shows the DFS up the control flow graph using the HASFATHER of each block.

```
PROCEDURE BLOCKDFS(STARTBLOCK)
BEGIN
STARTBLOCK^.MARKED:=TRUE;
BLOCKLIST:=STARTBLOCK^.HASFATHER
WHILE (BLOCKLIST <> NIL) DO
   BEGIN
   IF (NOT BLOCKLIST^.BLOCK^.MARKED) THEN
      BLOCKDFS(BLOCKLIST^.BLOCK);
   BLOCKLIST:=BLOCKLIST^.NEXT;
   END
END
```

Figure B.11: Pseudo code showing a depth first search of the basic blocks (traversing through each block just once in this case).

B.4.1 Pre- And Post- Dominator Blocks

The blocks immediately predominating and postdominating a given block can be extracted from the CFG as shown in Figure B.12, which are incorporated into the basic block data structure. Each block has its own unique immediate pre- and post- dominator blocks, which can be found within CAPTools by traversing the CFG from that block. A postdominator block will definitely be in all paths from a specified block to the routine end, whereas a predominator block will definitely be in all paths from the routine start to the specified block [118, 119]. A block can be pre- and post- dominated by many other blocks, but will only ever have one immediate pre- and post- dominator block. This immediate block can be used to traverse through all of the other pre- or post- dominator blocks (simplifying the pre- and post- dominator trees), i.e. all other dominators are found by traversing up the tree.

For example, in Figure B.12 Block 8 is immediately predominated by Block 5, where control must definitely have passed through Block 5 from the start node in order to reach Block 8. The reason why Block 6 and Block 7 do not predominate Block 8 is because the flow of control can pass through either of these to reach Block 8. I.e. Block 5 predominates Block 8 because the flow of control must definitely pass through Block 5 to reach Block 8, as no other route exists. Similarly, Block 8 is postdominated by Block 12, where no other control

flow path to the routine end exists from Block 8 that does not pass through Block 12 (the flow of control will pass through either Block 9 or Block 10).

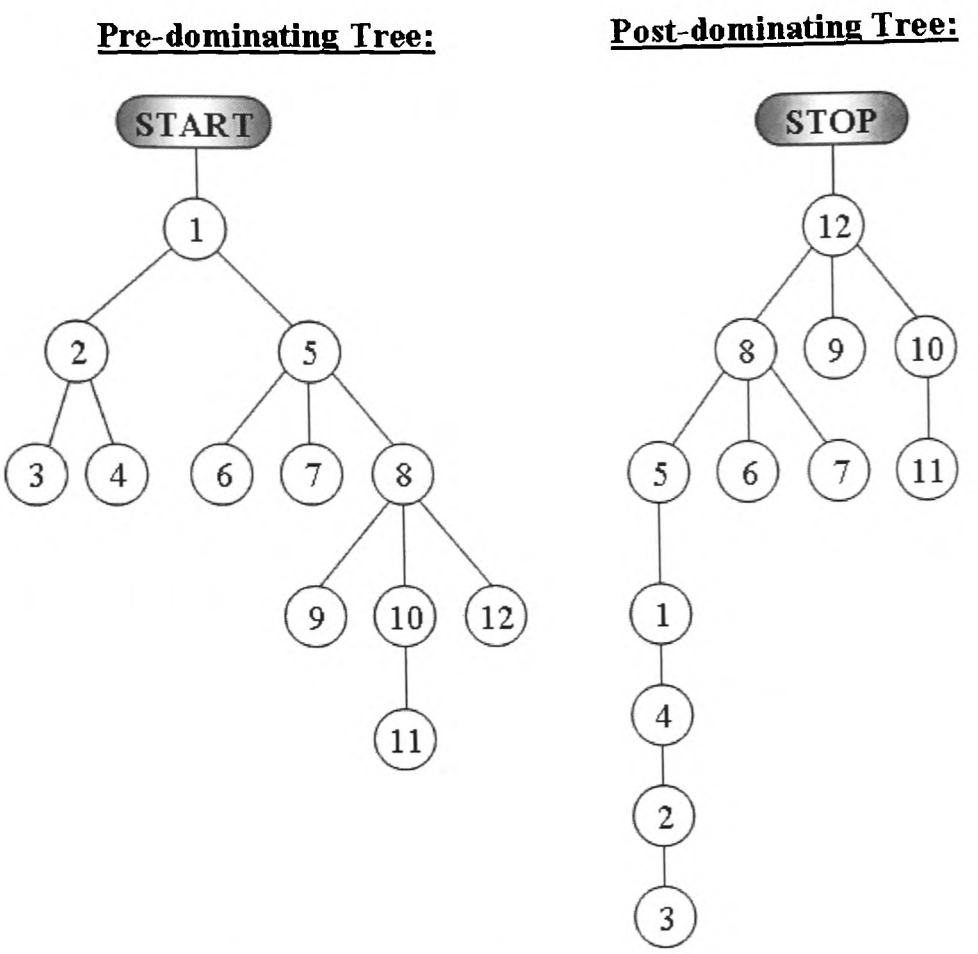


Figure B.12: Predominator tree and Postdominator tree for the CFG in Figure B.9, where each block has one immediate predominator and postdominator block. All other dominators are found by traversing up the tree.

The following pseudo code in Figure B.13 demonstrates a traversal up the predominator tree using these graphs and data structures. Therefore, using the example given in Figure B.8, if BLOCK is set to Block 11 then the code will traverse through the predominator graph passing through Block 10, Block 8 and Block 5 before reaching Block 1.

```
PREDOMBLOCK:=BLOCK;
WHILE (PREDOMBLOCK <> NIL) DO
  BEGIN
  ...
  PREDOMBLOCK:=PREDOMBLOCK^.PREDOM;
  END;
```

Figure B.13: Pseudo code used to traverse up the predominator graph within CAPTools.

The predominator tree can be used, for example, when migrating communications up through the code, allowing the communication to be executed

as early on as possible (see Section B.9 for more detail). The communication must definitely be executed before the usage of that data, which means that if placed in a predominator block of the usage then this will be guaranteed. The postdominator tree is used in the control dependence calculation (as discussed in Section B.6.2), where there is no control dependence to a postdominating block, since control will always flow to that block.

B.5 Nesting Information

When a block of statements is surrounded by a loop, then it is important for each of the statements to know the value of the loop variables being used and the number of iterations for each surrounding loop. For example, in Figure B.8 both statements S17 and S18 (Block 11) are contained within the loop headed by S16 (Loop S16 in the example), where both statements need to know the value of J and the loop limits (1, and NJ-1). Rather than storing this loop information for each statement within the block, the loop information can be stored for the entire block contained within the loop, reducing the amount of information stored within CAPTools. Block 11 only involves one loop, whereas Block 3 is contained within several loops (two in this instance), therefore knowledge of all of the surrounding loops is needed for each block. All of the blocks within a loop will store the same information pertaining to that particular loop, where for example Block 2, Block 3, and Block4, will store the details relating to Loop S1. The innermost loop is stored last in the list of nested loops and so Block 3 will first store information about Loop S1, followed by information about Loop S2. When a DO Loop is contained within an iterative loop (e.g. IF – GOTO), such as the first example shown in Figure B.14, then the iterative loop is at the outer nesting level, and is therefore stored at the beginning of the nesting list. The iterative loop is also at the outermost level in the second example in Figure B.14, where the DO 20 loop is restarted from I=1 when the IF condition (C1) is True, otherwise if the condition is False then another iteration of the DO 20 loop is performed.

```
10    DO 20 I=1,NI                10    DO 20 I=1,NI
        A(I)=                               A(I)=
        B(I)=                               B(I)=
20    CONTINUE                            IF (C1) GOTO 10
      ...                                 ...
      IF (C1) GOTO 10             20    CONTINUE
```

Figure B.14: Code demonstrating that the outer loop is the iterative loop and the innermost loop is the I Loop.

This NESTING information is stored within CAPTools for each block, where information relating to the surrounding loop (LOOPINFO) and a pointer to the next NESTING is given. The loop limits (LOWTREE and HIGHTREE) and the loop step (STEPTREE), as well as the pointer to the HEAD (the block that is the head of the loop), are stored within the LOOPINFO record only once for each loop.

B.6 Dependence Analysis

Dependence analysis is the most fundamental component of CAPTools, indicating data flow and memory re-use within a code [120, 121]. The various dependencies between the different statements within the code can be identified, enabling CAPTools to perform a good parallelisation. A miscalculation at this stage is critical, as a poor dependence analysis could lead to an unsatisfactory or incorrect parallelisation of the user's code. Dependence analysis exhibits all of the restrictions in execution order, where any generated code that executes a sink statement of a dependence before the source is invalid. CAPTools employs a conservative approach [24], where dependencies are assumed to exist unless proved otherwise. The powerful dependence analysis within CAPTools is used to detect possible parallelism throughout the code, and can be used to symbolically disprove many data dependencies that could lead to poor parallel performance. The dependence analysis identifies all of the dependencies within the user's code, indicating which sections of the code can be executed in parallel, and which must be executed in serial. This stage is very important as it is also used in deciding where to place execution control statements (masks) and communication calls (see Sections B.8 and B.9 respectively).

B.6.1 Dependence Types

A dependence refers to the relationship between two statements within a code, of which there are four basic types of dependencies [120], described in terms of a source and a sink. A True Dependence arises when the data from an assignment statement (source) is then used in a usage statement (sink). The source must obviously be executed before the sink statement. Consider Figure B.15a where a true dependence exists between statement S1, which assigns the value of T, and statement S2, which uses the data. A true dependence can also be marked as exact when every memory location accessed in the sink reference is also accessed in the source, which means that an exact true dependence represents a dependence where all of the data used in the sink is assigned in the source.

When the data is being reassigned after the usage of the same data then this is known as an Anti Dependence. The statement that uses the data (source) must therefore occur before the statement that reassigns that data, as the source is in effect overwritten by the sink. An anti dependence can be seen in Figure B.15b where the data used in statement S1 is reassigned in statement S2.

An Output Dependence occurs when data is being reassigned after being previously assigned. This is a common method used in many codes to reuse memory locations with the aim of reducing memory overheads. From Figure B.15c it can be seen that the data in statement S1 is simply reassigned in statement S2.

Finally, when a control statement, such as an IF, controls the execution of other statements, as seen in Figure B.15d where statement S2 is controlled by statement S1, then this is known as a Control Dependence. In this case the statement S2 may not execute until statement S1 has been proved either true or false.

True Dependence	Data used after assignment
Anti Dependence	Data reassigned after usage
Output Dependence	Data reassigned after previous assignment (reuse)
Control Dependence	Statement controls execution of another statement

a - TRUE	S1	T(I)=...	**b - ANTI**	S1	...=T(I)
	S2	...=T(I)		S2	T(I)=...
c - OUTPUT	S1	T(I)=...	**d - CONTROL**	S1	IF (conditional) THEN
	S2	T(I)=...		S2	T(I)=...

Figure B.15: The different types of dependencies. a:-true dependence; b:-anti dependence; c:-output dependence; and d:-control dependence.

B.6.2 Control Dependence Calculation

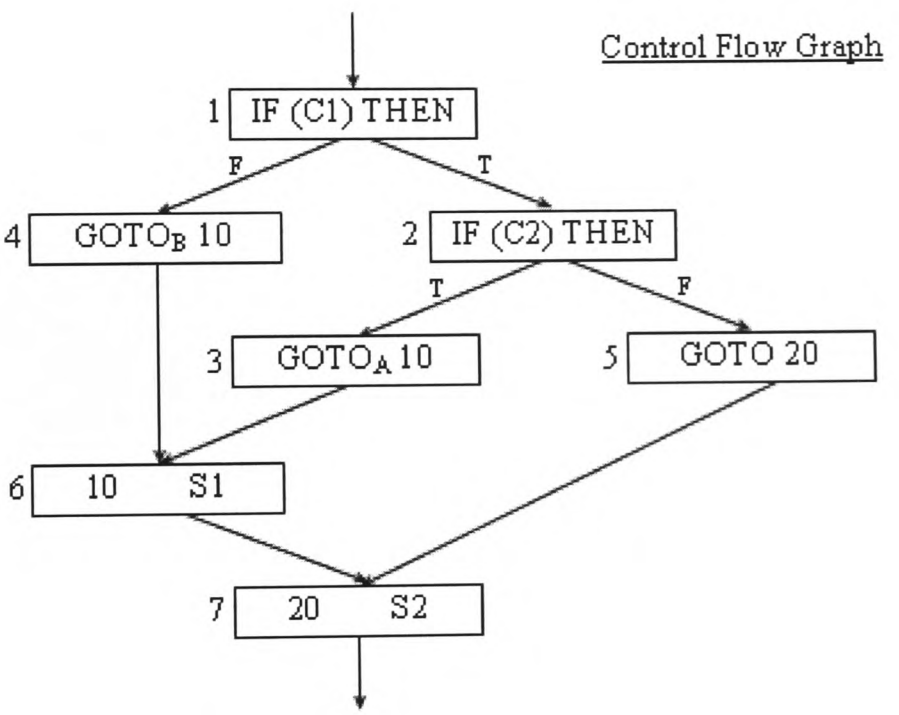
Prior to full dependence analysis, control dependencies are calculated. The algorithm sets the dependencies within the code [119], which are used transitively to give full control. For example, S1 in Figure B.16 will be executed either when both C1 and C2 are true, or when C1 is false.

The execution of S1 will be dependent upon the control values of C1 and C2, where it is unknown at compile time what these values will be. If the condition C1 is true then control will flow to the next IF statement involving C2, otherwise if C1 is false then the "$GOTO_B$ 10" statement will be executed. Similarly, when the condition C2 is true then the "$GOTO_A$ 10" statement will be executed, leading to the execution of S1 (the statement in question), otherwise if C2 is false then the "GOTO 20" statement will be executed, meaning that S1 will be skipped over. Therefore S1 is control dependent on the values of C1 and C2, where this information can then be used whenever S1 is involved.

Using the postdominator tree that has been constructed from the CFG, the control dependencies are calculated in order to determine whether a statement will execute. If a statement (or block of statements) does not postdominate its father statements, then it is said to be control dependent on those father statements [119]. The control dependence calculation algorithm searches up the postdominator graph until a common postdominator is reached. All of the statements (blocks) that were traversed then contain statements (blocks) that are control dependent on the father block.

The pseudo algorithm that is used by CAPTools for control dependence calculation is given in Figure B.17, which is applied to the example given in Figure B.16.

Block:		*Code:*
1		IF (C1) THEN
2		IF (C2) THEN
3		$GOTO_A$ 10
		END IF
		ELSE
4		$GOTO_B$ 10
		END IF
5		GOTO 20
6	10	S1
7	20	S2



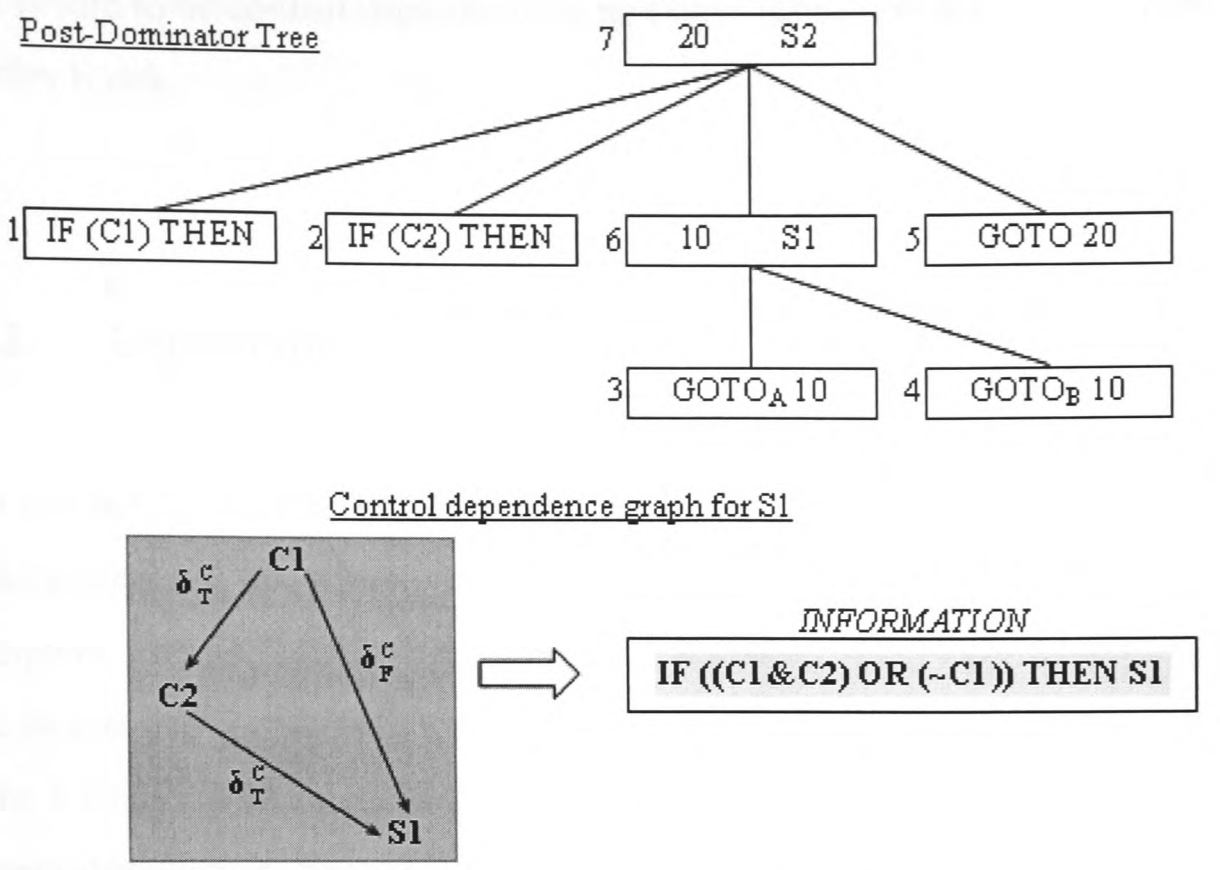


Figure B.16: Sample code, with its control flow graph, postdominator tree, and the control dependence graph for S1, which illustrates that S1 is dependent on C2 being True, given that C1 was previously True, OR that C1 was False.

Pseudo Algorithm:

For each child block (where more than 1 exists):
- Mark all post dominators of father
- Set all post dominators from child block until marked block as being control dependent on father

Control dependence calculation for S1:

For block 1:
child 4 - blocks 4, 6, control dependent on block 1 (from postdominator tree)
child 2 - block 2 control dependent on block 1
For block 2:
child 3 - blocks 3, 6, control dependent on block 1
child 5 - block 5 control dependent on block 1

Figure B.17: Pseudo algorithm used by CAPTools for control dependence calculation. Also shown is the application of this algorithm on the calculation of the control dependence graph for S1 in the example given in Figure B.16.

Block 1 has more than one exit and is postdominated by Block 7 which is therefore marked. The child blocks of Block 1 are examined (Block 4 and 2), where these child blocks and any of their postdominators other than the marked block are all said to be control dependent on Block 1. A similar process is undertaken for Block 2 which also has more than one exit. In general, a child

block is said to be control dependent on its father block if it does not postdominate its father block.

B.6.3 Dependence Depth

Loop carried dependencies [120, 121] are another attribute to consider, as these are dependencies for which the source is in one iteration and the sink is in a subsequent iteration of the same loop. To represent this, each dependence also possesses a depth, where a dependence may be Loop Independent if it exists within a single iteration of all surrounding loops, or Loop Dependent if it exists between iterations of any of the surrounding loops. For example, in Figure B.18 the value of A(I,J) was assigned and used in the same iteration of both the I and J loops, implying that there are no loop carried dependencies in this loop.

Independent

```
DO I=1,NI
  DO J=1,NJ
    A(I,J)=...
    ...=A(I,J)
  END DO
END DO
```

Figure B.18: Example of a loop independent code, in which data is respectively assigned and used in the same iteration of the I and J loop.

The level of dependence can be determined by examining the dependence between the different levels of loops surrounding the statement(s). For example, if the values used in an iteration were assigned during earlier iterations of the outermost loop then it is deemed to be a level 1 dependence, as demonstrated in Figure B.19 where the data that was assigned in a previous iteration (2 iterations before) of the K loop is used. If a dependence exists between iterations of the next outermost loop of the surrounding statement(s) then it is deemed to be level 2, as demonstrated in Figure B.20 where the data used was assigned in the previous iteration of J. If an assigned value of A was required from an earlier I iteration, then this would be deemed a level 3 dependence (which in this instance would be

the innermost loop). Therefore the level of dependence is determined by the level of the nested loop in which the data value is dependent upon.

Level One Dependence

```
DO K=3,NK
  DO J=1,NJ
    DO I=1,NI
      IF (I.NE.J) THEN
        A(I,J,K)=A(I,J,K-2)
      END IF
    END DO
  END DO
END DO
```

Figure B.19: A level 1 dependence, where the usage of A was assigned during an earlier iteration of the outermost loop (K).

Level Two Dependence

```
DO K=1,NK
  DO J=2,NJ
    DO I=1,NI
      IF (I.NE.K) THEN
        A(I,J,K)=A(I,J-1,K)
      END IF
    END DO
  END DO
END DO
```

Figure B.20: A level 2 dependence, where the usage of A was assigned in the previous iteration of the J loop.

B.6.4 Loop Normalisation And Induction Variable Substitution

Loop normalisation [122] consists of transforming existing DO Loops using the transformation shown in Figure B.21, so that the loops start from 1 and increment in steps of 1. An example of the normalisation of a loop that starts from 3 and has an incremental step of 2 is illustrated in Figure B.22. Induction variables, which have constant increments in every iteration of a particular loop, are identified and transformed to be functions of the loop variable concerned [121, 122, 123]. These transformations are not essential but they do simplify the analysis and the code generation stages, where these transformations are easily reversible during the code generation stage to ensure original code recognition [25]. Normalisation

essentially means that anything involving the normalised loop can be easily tested, as all loops start from 1, and have an incremental step length of 1.

Transformation used to Normalise a loop

```
DO I=L,H,S
   A(I)=
END DO
```

```
DO i=1,(H-L)/S+1,1
   A((i-1)*S+L)=
END DO
```

Figure B.21: Transformation used to Normalise a loop, where the loop starts from L, ends at H, and has a step length of S.

Un-normalised

```
DO I=3,NI,2
   A(I)=A(I-1)
END DO
```

Normalised

```
DO i=1,((NI-3)/2)+1,1
   A(2i+1)=A(2i)
END DO
```

Figure B.22: An un-normalised loop (starting at 3 and with a step length of 2), with the normalised version of the same loop (starting from 1 and with a step length of 1).

B.6.5 Dependence Testing

Various information is used in the dependence tests, such as the loop nesting, control statements (IF's, computed GOTO's, etc), index equality (for arrays), and user volunteered information (e.g. READ variables). For example, the values of NI, NJ, and NK, may be read in at runtime, and so the values of these variables are not known (by a parallelising compiler) during the dependence analysis stage of the parallelisation unless the user submits this information.

The example shown in Figure B.20 can be examined for any dependencies, testing whether any dependencies exist between iterations of any of the loops. For simplicity, this is demonstrated for a True dependence, although it also applies to Anti and Output dependencies. The main statement in Figure B.20 can be expressed as that given in Figure B.23, where I_a gives the value of I in an assignment, and I_u is the value of I in a usage. Similarly, J_a and J_u are used to represent the value of J in an assignment and usage respectively, and likewise for K_a and K_u for the value of the K in an assignment and usage. For a dependence to exist, the constraints of Figure B.20, shown in Table B.1, must be satisfied.

```
DO K=1,NK
  DO J=2,NJ
    DO I=1,NI
      IF (I.NE.K) THEN
        A(Ia,Ja,Ka)=A(Iu,Ju-1,Ku)
      END IF
    END DO
  END DO
END DO
```

Figure B.23: Example used to demonstrate dependence testing, where X_a is the value of index X in an assignment, and X_u is the value of index X in a usage, from which the constraints can be constructed (shown in Table B.1).

The lower and upper constraints for each of the loop variables can be extracted from the loop nesting information, where the loop variable values of a particular loop must both lie between the given limits of that loop. For example, the value of A can only be assigned and used for values of K between 1 and NK, similarly for values of J between 2 and NJ. When setting up the dependence depth constraints, it is assumed that the value of the loop variable in the assignment is less than the value of the loop variable in the usage, such that the assignment is made in a previous iteration to the usage (as this example tests for a True dependence). With the level 1 dependence, there are no surrounding loops, and so for there to be a dependence K_a would have to be less than K_u. When looking at the level 2 test information for the J loop, we are in a single iteration of the surrounding loops, and so $K_a=K_u$. However, for there to be a dependence between the assignment and usage of A (on this particular statement in this instance), then the assignment in the J index will have had to have been assigned before the usage in the J index ($J_a<J_u$). Similarly for the level 3 test information for the I loop, both the K and J loop variables will be constant, where the assigned value of A in the I index may only be used in subsequent iterations of that I loop. Following this trend, the loop independent test information can be set up (for independent loops), where the I, J, and K, loop variables are all constant.

The index equality constraints can be set up using the assertion that the memory location of the assignment of A is the same as the memory location of the usage of A. There is a dependence between the assignment and usage if the memory locations of these indices could be the same. The control information can be obtained from the condition inside the IF statement (I.NE.K), which implies that the assignment of A only occurs when I_a is not equal to K_a. Similarly, A will

not be used when I_u is equal to K_u. This is identified using the control dependencies (Section B.6.2) to identify a comprehensive control set for the references.

Constraints						
From Nesting:	**Level 1**	**Level 2**	**Level 3**	**Level Infinity (Independent)**	**Index Equality**	**Control**
$1<=K_a<=NK$	$K_a<K_u$	$K_a=K_u$	$K_a=K_u$	$K_a=K_u$	$K_a=K_u$	$I_a<>K_a$
$1<=K_u<=NK$						$I_u<>K_u$
$2<=J_a<=NJ$		$J_a<J_u$	$J_a=J_u$	$J_a=J_u$	$J_a=J_u-1$	
$2<=J_u<=NJ$						
$1<=I_a<=NI$			$I_a<I_u$	$I_a=I_u$	$I_a=I_u$	
$1<=I_u<=NI$						

Table B.1: Loop constraints used in disproving assumed dependencies for Figure B.20.

If there is a contradiction between the various constraints then the dependence is proved non-existent, otherwise the dependence is assumed to exist. For instance, in Table B.1 the index equality constraint ($K_a=K_u$) contradicts the dependence depth constraint at level 1 ($K_a<K_u$), and so there is no dependence between the different iterations of the K loop. When comparing the index equality constraints ($J_a=J_u-1$) against the level 2 constraints ($J_a<J_u$) there is no contradiction, implying that this assumed dependence can definitely not be removed. At level 3 there is a contradiction since $J_a=J_u$ and $J_a=J_u-1$ do not match and because and there is also a conflict between $I_a<I_u$ and $I_a=I_u$, implying no dependence. Similarly, there is no dependence at level infinity due to conflicting constraints.

In Figure B.24 a common surrounding loop is used around two independent loops, where A is assigned in the first inner loop and used in the second inner loop. The value of K remains constant for each iteration, and so there is no dependence between successive iterations of the K loop. The second J loop uses values of A that have already been assigned in the first J loop, and so there are no loop carried dependencies between these loops. Additionally, there is no level 2 test since there is only one common loop, although a loop independent test can be used to examine the dependence between the assignment of A in the first J loop, and the usage of A in the second J loop, in which K is constant inside the two independent loops.

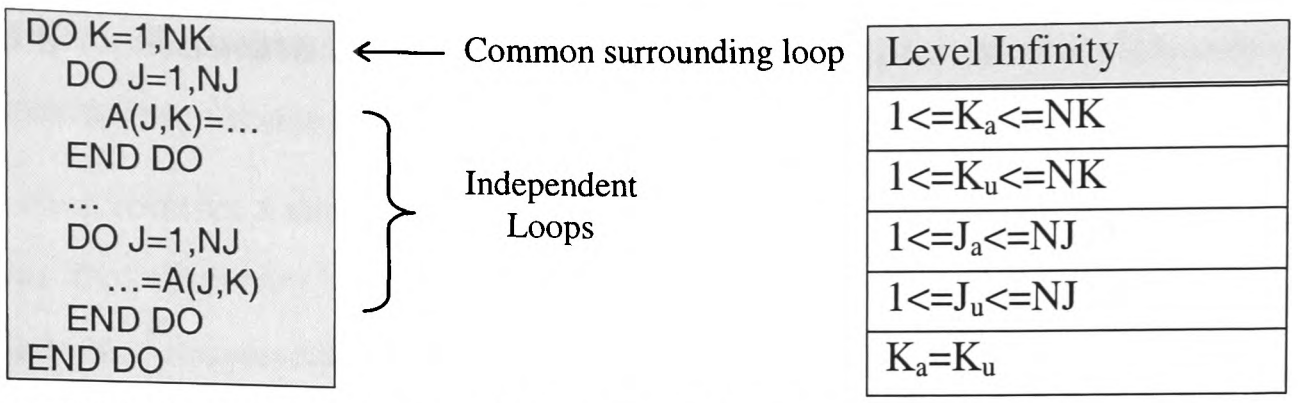


Figure B.24: Example of Level Infinity constraints for two independent loops surrounded by a common loop.

The Greatest Common Divisor Test (GCD) [121], the Banerjee Inequality Test [121, 124, 125], the Symbolic Inequality Disproof Algorithm (SIDA) [24, 111], and the Omega Test [126], all use available information to test the non-existence of data dependencies in the code. An inference engine [127] is also used, where inferred knowledge can be used in these tests such as that in Figure B.16 (involving AND and OR operators). Logical substitution [24] is used when several definitions of a variable exist, as demonstrated in Figure B.25 where two tests will be performed when testing A(K). Both tests must be proved false in order to prove that a dependence does not exist.

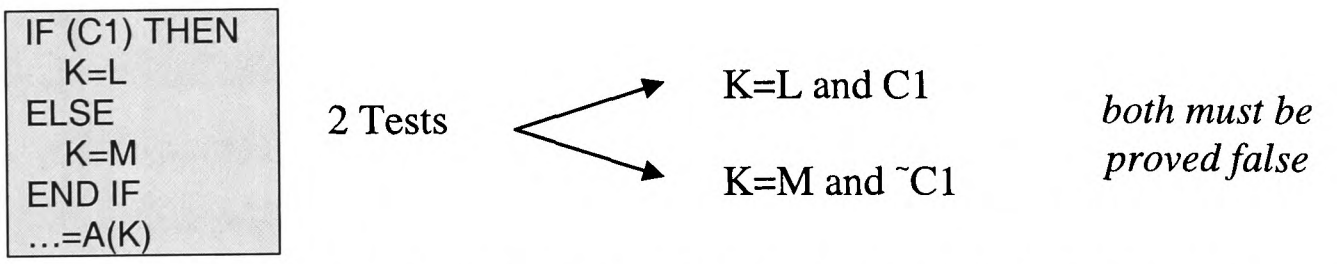


Figure B.25: Example where the inference engine and logical substitution is used in dependence testing, where both values of K must be proved false for any test.

CAPTools uses loop carried dependencies to detect serial loops, in which pipelines [111] are sometimes generated due to the use of data calculated in previous iterations. The dependencies for each executable statement are stored within the COMMAND data structure of CAPTools, where each dependence data structure stores the information for its depth, type, and the variable that causes that dependence.

B.6.6 Routine Dependence Graph

For each routine, a dependence graph [111, 120, 121] is constructed, consisting of nodes that represent executable statements, and directed edges that flow from node to node representing the dependencies between the statements (Figure B.26). In the example there are 5 True dependencies and 9 Routine Input/Output dependencies relating to the variable PHI in the routine LISOLV. A basic dependence calculation is performed which consists of a scalar and array analysis, where a scalar variable can be a DO Loop counter variable for example whose value will always be defined within the loop.

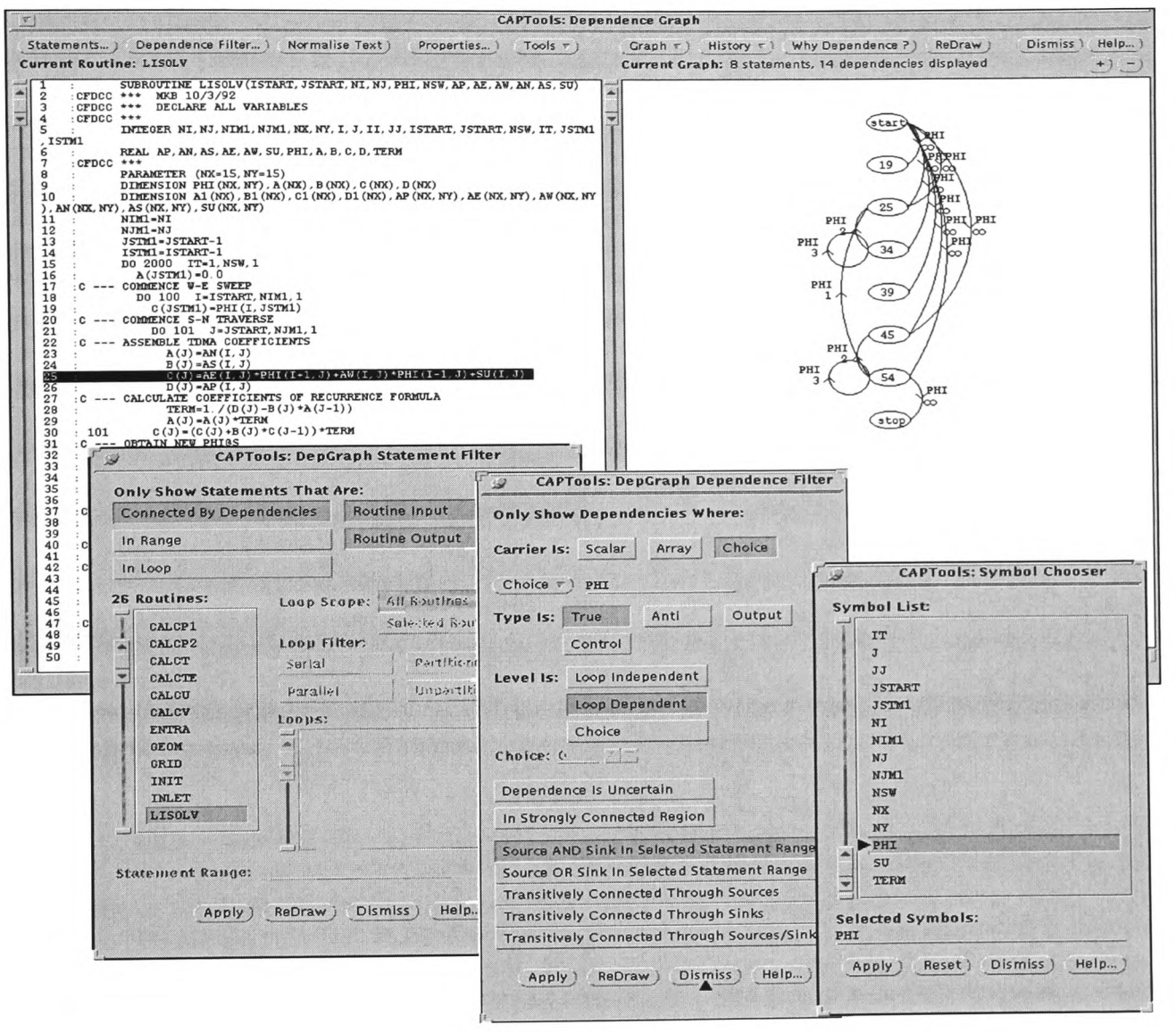


Figure B.26: Example of the Dependence Graph window within CAPTools, along with the Statement and Dependence Filter options selected (from which the user can select which dependencies to view).

B.6.7 Interprocedural Dependence Analysis (Routine Input And Output)

To accurately and comprehensively represent dependencies in the entire application code being analysed, interprocedural analysis is essential. A routine may have several parameters or commons, some of which are passed in (Routine Input), and some of which are calculated and passed out to the calling routine (Routine Output). After a routine dependence graph has been constructed the START and STOP nodes of that routine are added, which are used in the passing of data between routines that are connected in the call graph. All statements within a routine that use variables that have not been defined in the routine but that have been passed in via either the parameter list or common block are linked to the start node in the dependence graph as demonstrated in Figure B.27. Similarly, any statements that define variables that are passed out of the routine are connected to the stop node. Therefore dependencies not only exist between various statements within a particular routine, but they can also exist between routines. This also includes SAVE statement variables and also any local variables in a routine that may use uninitialised values. These are stored in common data structures and are inherited by caller routines.

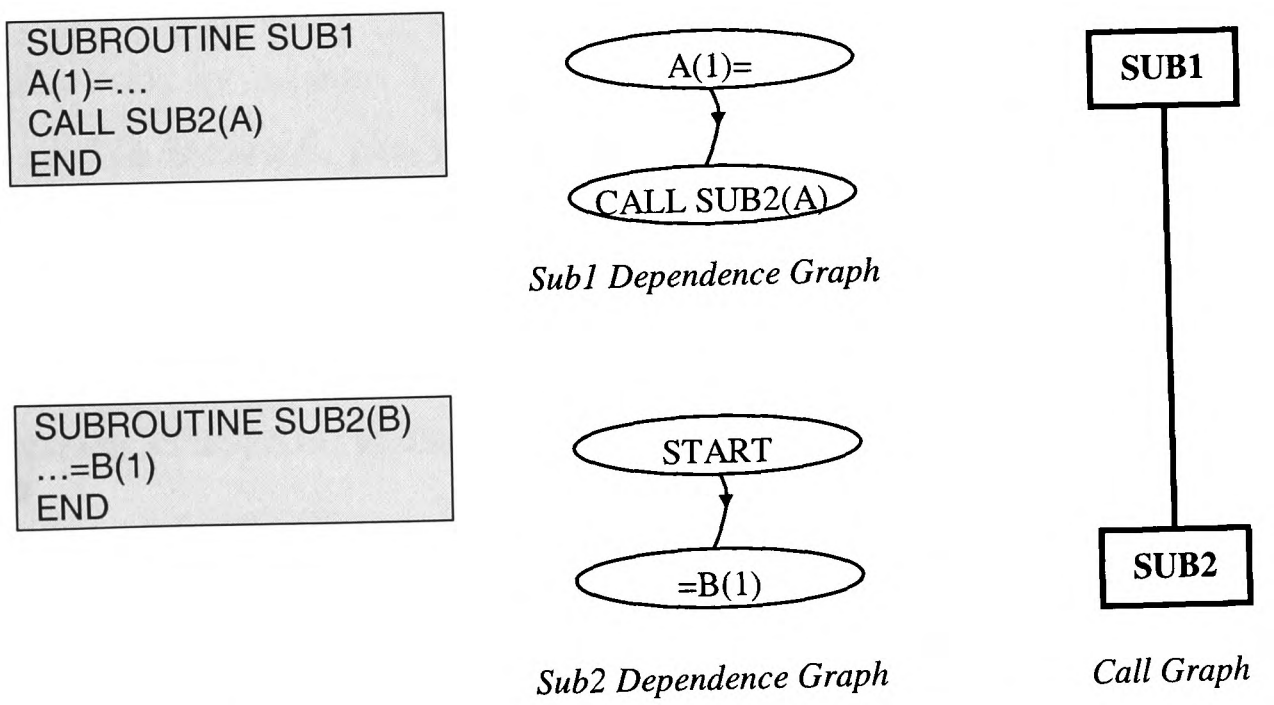


Figure B.27: Sample code in which A is assigned in the calling routine and used in the called routine (as B).

An atomic dependence test [24, 111] checks all references in called routines, and maps into the original routine (i.e. array dimensions). The test uses control on all call sites and variable references, where any reference that sets a dependence terminates the test. Array sections [24] are used to summarise all of the references in a routine for a given array, allowing pre-tests to avoid the worst case (1000's of atomic tests with no dependencies).

B.6.8 Value Based Covering Sets

For an effective parallelisation, the analysis must be value based, i.e. flow of data only for true dependencies and not memory re-use. Covering sets are used when trying to determine possible dependencies between certain assignment statements and usage statements. For example, in Figure B.28 there is clearly a dependence between S2 and S5, but is there a dependence between S2 and S13? It is assumed that a dependence between these two statements exists until proven otherwise, and so covering sets can be used when trying to determine the possible dependencies between the assignment and usage statements.

If any of the assigned data in Section 1 is used in Section 6 then a dependence exists. However, the entire usage range of Section 6 is assigned collectively by Sections 3, 4, and 5, where none of the data assigned in Section 1 is used in Section 6. This means that the dependence between S2 and S13 can be disproved since the entire usage range (I=1,NI) is covered by the assignment range of Sections 3, 4, and 5 (I=1, I=2,NI-1, and I=NI, respectively). If, for instance, A(1) was not reassigned in S7 then there would be a dependence between S2 and S13, as the entire usage range of Section 6 would not be covered.

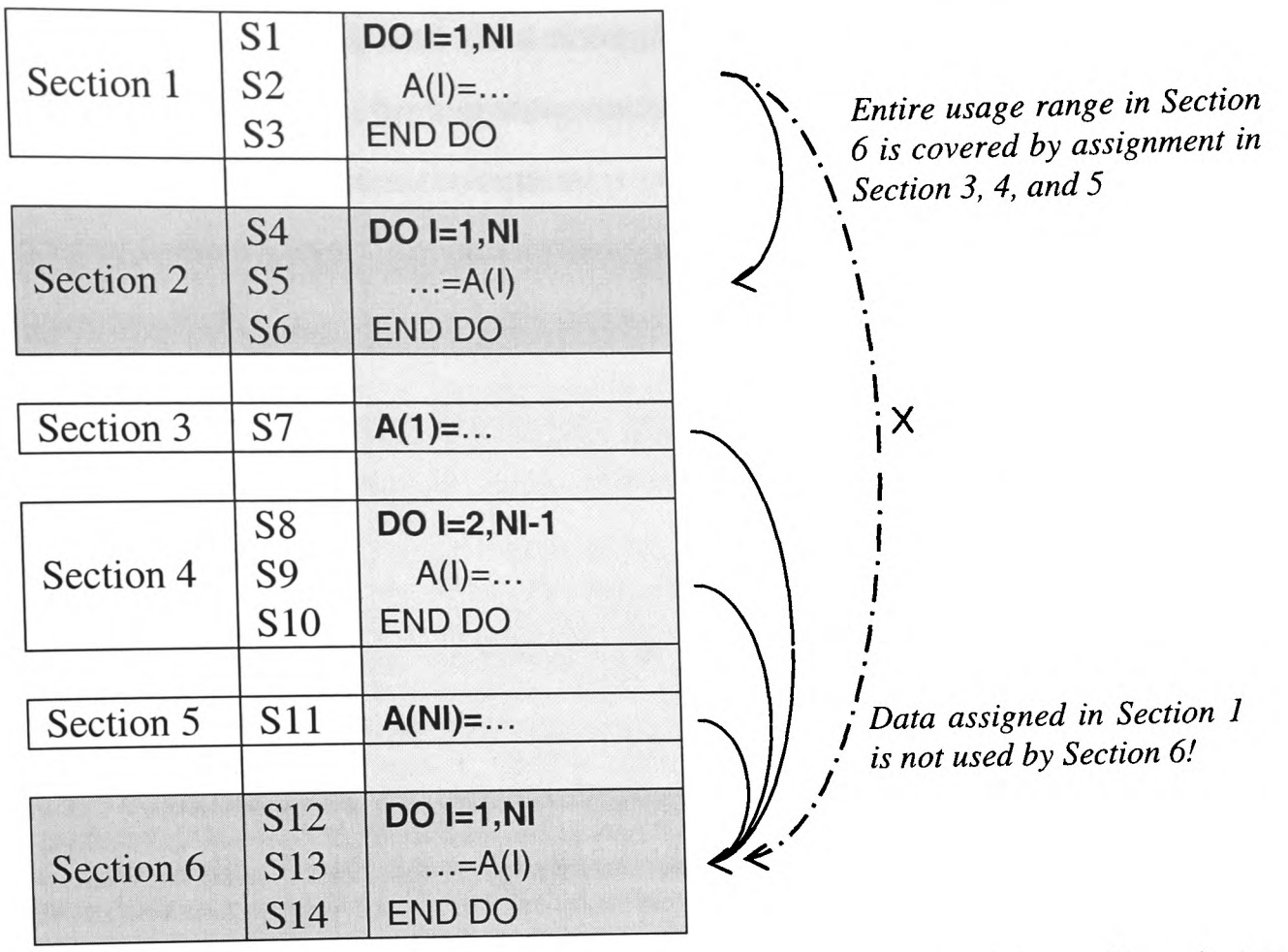


Figure B.28: Example showing that the usage of A in Section 6 is not dependent upon the assignment of A in Section 1, as all of the usage range has been assigned in Sections 3, 4, and 5.

B.6.9 User Interaction In Dependence Analysis

To ensure that the dependence analysis obtained is as accurate as possible it is vital that there is some form of user interaction. The user could, prior to the dependence analysis, submit additional information, as this may aid in minimising possible control flow paths and also remove any dependencies that would otherwise have been assumed to exist. For example, the user could supply beneficial knowledge relating to variables that are read in at runtime, or about other variables used in the code. The user is also able to answer frequently asked questions that are asked by CAPTools during analysis, which is useful if the user did not know that this information was important prior to analysis. The user can also query dependencies, deciding whether or not they should exist, which can be performed. User knowledge is of great importance in producing an efficient parallel code, as the user's knowledge can be used to remove serialising dependencies that could not be disproved without this knowledge. Only the user

has this knowledge, and so a compiler would definitely not be able to remove these dependencies, highlighting the benefit of using an interactive parallelisation tool over a parallelising compiler.

B.6.10 Symbolic Variable Manipulation

In order to obtain an accurate dependence analysis it is essential to be able to manipulate symbolic variables [111]. A symbolic variable must therefore be precise, otherwise a poor (or incorrect) dependence analysis could be performed.

A variable can be defined either as a loop variable, or as a nonloop variable (e.g. I and NI respectively in Figure B.28). Nonloop variables are often found in index expressions as constants or coefficients of loop variables, in loop limits, and in conditional statements. Not only are they defined in terms of the symbol of the variable but also as the defining statement of the variable along with the call path to the routine that assigns the variable, enabling a more accurate comparison of these nonloop variables.

B.6.10.1 Symbolic Variable Equality

It is important to be able to identify unique instances in which data has been defined, allowing the user to correctly trace any variable through the code, which can only be achieved when using the call path. For example, in Figure B.4 it can be proved using the call paths that the value of t is not necessarily equal to the value of n in Main (i.e. does t=n?). The variable t can be traced down through the code to have the value r (read in externally), as can n. However, the value of r can be different for each instance of the called Sub3, as the entered value depends on the user. Although both t and n are read in as the variable r, these are different instances, and so it would not be possible to say that t is equal to n all of the time.

For example, in Figure B.4 the value of c can be traced up through the call path of the code, where the value of c is a particular instance of the entered r value

in Sub3, as illustrated in Table B.2. So in the usage statement of c (S_9), c can be traced up to the calling statement in the Main program (S_3), where the value of n can be traced up further to the caller statement of Sub1 (S_2). The variable n is passed into Sub1 as x, which is used in the call to Sub3 (S_5), which is passed into Sub3 as r. The variable r is read into the code (assigned) in statement S_7, and so the unique path of this data is Sub1.S_5 → Main.S_2.

Tree	Command	Routine	Call Path
c	S_9	Sub2	
n	S_3	Main	
n	S_2	Main	
x	S_5	Sub1	Main.S_2
r	S_7	Sub3	Sub1.S_5 → Main.S_2

Table B.2: Call path for the usage of c in Sub2 traced to definition, for the example shown in Figure B.4.

B.6.10.2 Using Symbolic Variables

Symbolic variable lists are used to store array reference information (in the STATEMENT record type), as shown in Figure B.29. The symbol T is stored in the SYMB field of the STATEMENT data structure, where the expression list of the indices of this variable (if it is an array) is pointed to by the LINK field. The EXPRESS record stores information for a symbolic expression (e.g. an array index expression), where the COEF field points to a list of loop variable coefficients, and the NONLOOP field points to a list of nonloop variables. The CONSTANT field holds the integer constant component of the expression, and the NEXT field points to the next symbolic expression record in a list (e.g. the next array index). For example, for the assignment of T in Figure B.29 the integer part of the loop variable coefficient for the J and K loop variables is 0, and the coefficient for the I loop variable is 1, where the LOOPINFO pointer points to the loop information record. The NONLOOP record stores information relating to one or a product of symbolic variables, where the TERM field points to a list of individual symbolic variable instances where the overall expression is the product of that list. The COEF field is the integer coefficient of the symbolic variable list, and the NEXT

field points to the next nonloop variable record where entries in the list are summed to create an overall expression. In this example, COEF is set to 1, and NEXT is set to NIL (represented as an X in Figure B.29) since there are no more nonloop entries in this array index expression. The TERM record stores information that precisely defines a symbolic variable instance, defined as an individual nonloop variable. The TREE field points to the parse tree (also shown in Figure B.29) node that represents the reference to the variable in this instance, and the COMMAND field points to the command at which the value for this instance is defined (or can be found via dependence fathers). For example, in Figure B.29 TREE points to the parse tree node for N, and COMMAND points to the command that assigned this instance of N. The expression for the right-hand side usage of V is also shown in Figure B.29, where this expression consists of loop variables, nonloop variables, and constants, which can be stored in the same way. For example, the COEF field in the COEFFICIENT record for the I and K loop variables is 0, and is –3 for the J loop variable, and the symbolic NONLOOP part of the loop coefficient points to M. The NONLOOP part of the expression points to a specific instance of N, similar to that in the array index expression. The symbolic variable data structure may be manipulated using some of the utilities in the next Section.

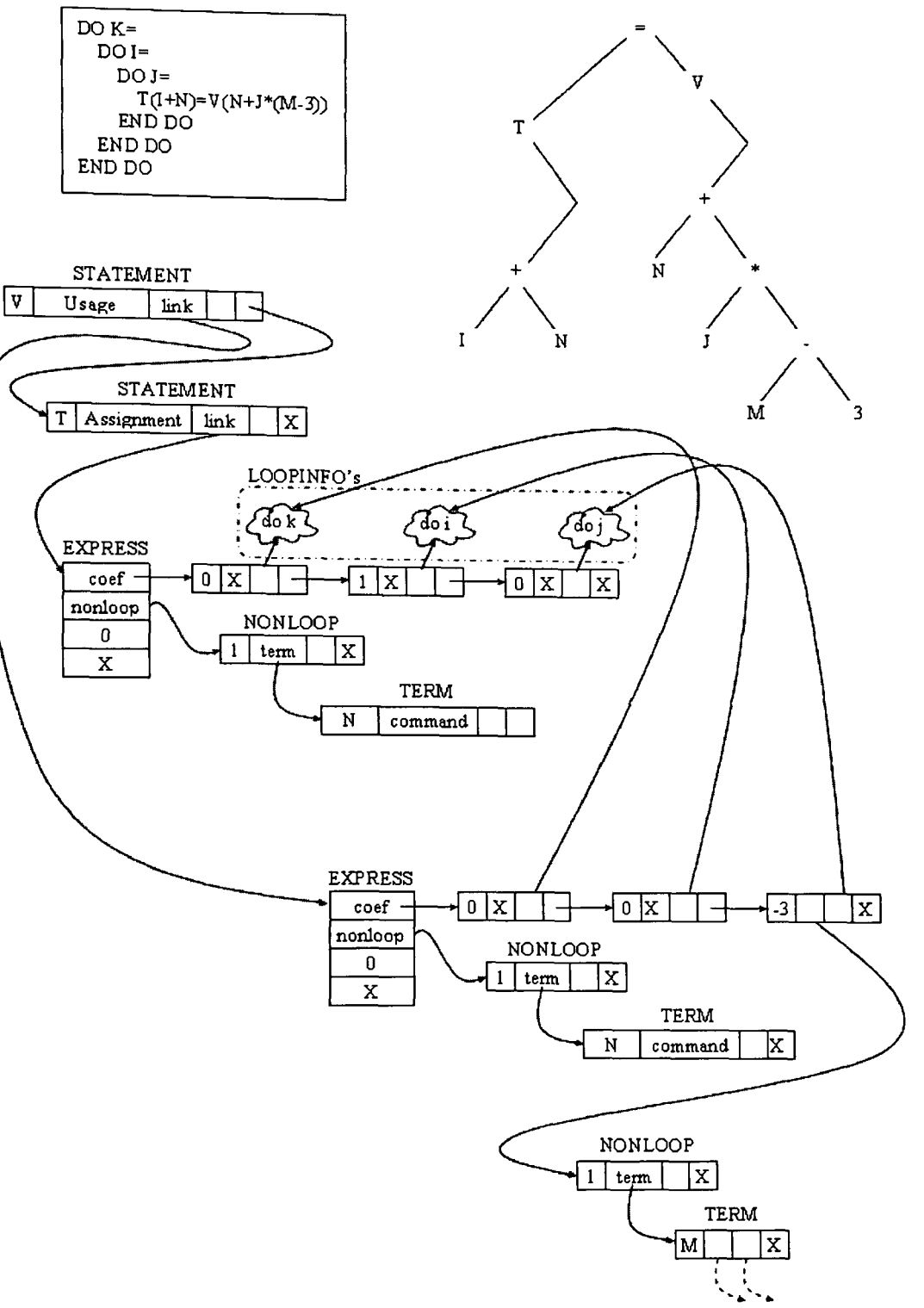


Figure B.29: The parse tree and the symbolic data structures that are associated with the given assignment statement. Note that X is used to represent a NIL entry.

B.6.10.3 Symbolic Variable Manipulation Utilities

Several utilities exist within CAPTools that can be used to manipulate these symbolic variables and their data structures, whereby the algorithms within CAPTools may be used to exploit the symbolic algebra, some of which are shown in Table B.3.

Utility Name:	Explanation:
FORSUBSTITUTE	Takes the parse tree, or an existing symbolic expression, and returns a substituted symbolic expression based on definers in the dependence graph
ADDLIST	Allows two symbolic variable lists to be added or subtracted
MULTLISTS	Allows two symbolic variable lists to be multiplied together
EXPRESSTONON	Converts a list of loop index coefficients and nonloop variables into a nonloop only list
EXTRACTLOOPS	Extracts loop variable coefficients from a nonloop list
OTHERNAME	Returns the name of the given variable in the called routine
CALLEDNAME	Returns the name of the given variable in the calling routine
LDISPROVE	Processes a given expression with comparison operator (i.e. $<$, $\leq$, $=$, $\geq$, $>$) and determines whether false or undecided
INFERENCE	Uses the inference engine to determine whether a control set can be proven false, otherwise it is undecided
SYMBOLICMOD	Calculates EXPRESSION1 mod EXPRESSION2
SYMBOLICDIV	Calculates EXPRESSION1 div EXPRESSION2
CREATEBLOCK	Creates a new block within the code, where this block can be placed either before or after a specified block, and at the same or at a lower loop nesting to that block
INLINECOMMONS	Will enable the common block from the Main routine to be placed into any other routine where variables are statically declared
BUILDTREE	Takes nonloops and builds a tree (the opposite of FORSUBSTITUTE)
ADDDECLCOMMAND	Add a command to the end of the declaration list

Table B.3: Table of symbolic variable manipulation utilities.

These routines form the building blocks for the algorithms implemented in CAPTools, including those that are used in Chapter 5.

B.7 Data Partitioning

The dependence graph exhibits the parallelism within the code, and so it must now be decided how to exploit this effectively. In terms of a single array, as shown in Figure B.30, a processor will operate on the entire array when executing in serial. However, when executing in parallel, a more efficient approach would be to partition the array across the processors such that each processor "owns" an allocated section of the array that they operate upon. Although each processor can operate on the entire array which they still own, they will usually only be able to assign data into their own subsection of the array. The performance of the parallel code can be enhanced by effectively partitioning the data, such that each processor can operate independently of one another as much as possible. Note that each processor executes a copy of the parallel program upon their data set using the SPMD paradigm on a DMS.

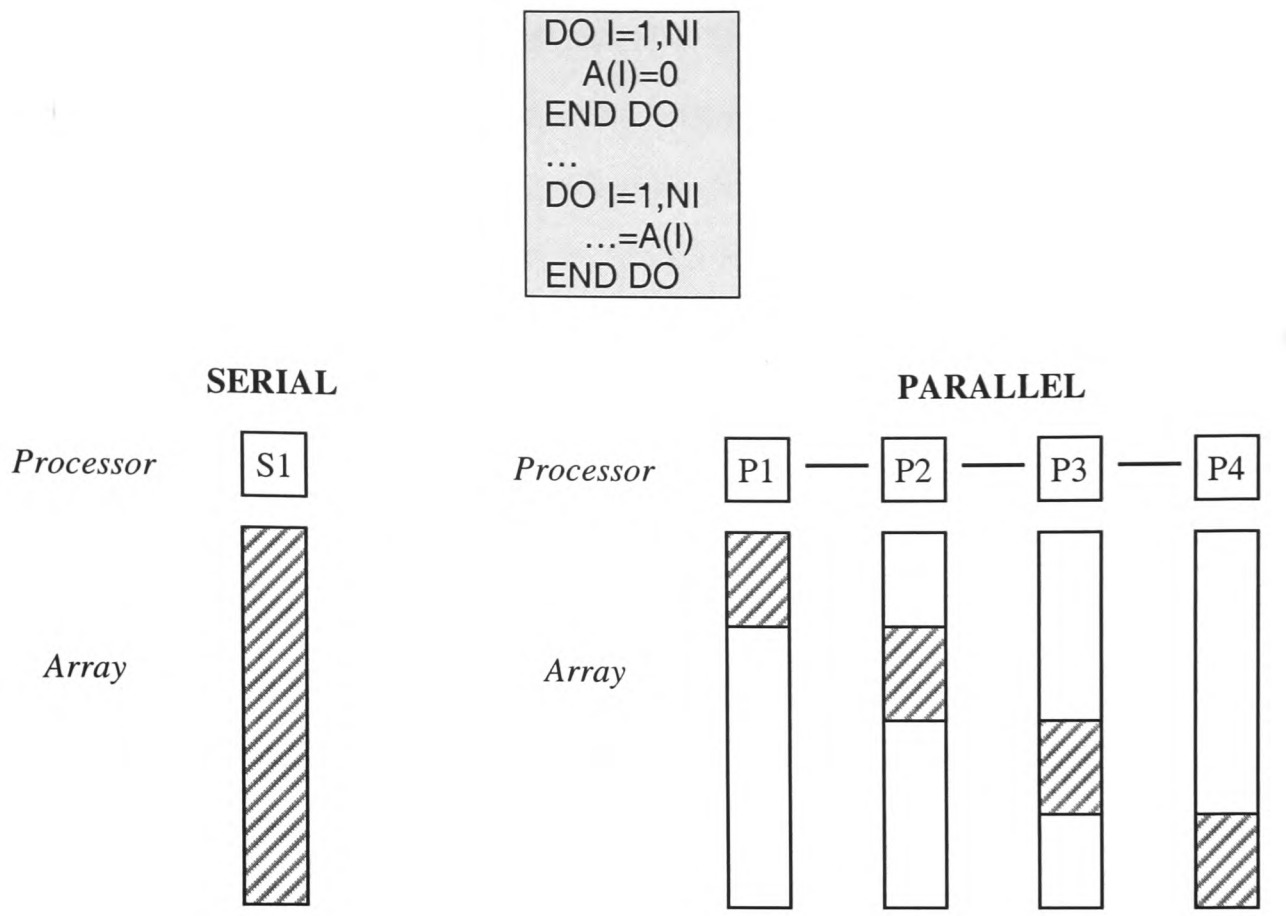


Figure B.30: Representation of an array that is initialised in serial, and in parallel (using 4 processors) where each processor operates upon the shaded region of the array.

Once an array has been partitioned then this particular partition must be used throughout the code, as it would be incorrect to use one partition to assign data, and a different partition to use the data. If a processor assigns data within its subsection then this is where the data is actually located, which means that if the partition is changed then the assigned data may no longer be available for use (as this is not within the new partition range). Consider the example code given in Figure B.30, where the array A is assigned and used between 1 and NI throughout the code, then it is possible to partition A across the processors as shown.

All of the relevant arrays for this memory area should be partitioned, which implies that arrays must be partitioned interprocedurally. For example, in Figure B.31 if A has been partitioned in Sub1, then since this workspace is used as X inside Sub2, then X must also be partitioned, and likewise for Y in the calling routine (Main).

PROGRAM MAIN	SUBROUTINE SUB1(A)	SUBROUTINE SUB2(X)
...	...	...
DO I=1,NI	DO I=1,NI	DO I=1,NI
Y(I)=...	A(I)=...	X(I)=...
END DO	END DO	END DO
...	...	...
CALL SUB1(Y)	CALL SUB2(A)	
...	...	
END	END	END

Figure B.31: Sample code demonstrating that both X and Y must be partitioned (interprocedurally) when A is partitioned.

Linear relationships between array indices infer partitions to other arrays (i.e. it is desirable to align related arrays to the processor topology). Most application codes do not simply deal with one-dimensional arrays, but with multi-dimensional arrays, and so it is necessary to be able to partition these arrays too. In Figure B.32 for example, if A is partitioned in index 3, then the statement implies that array B should be partitioned in index 2 since a linear relationship exists involving K (a loop variable).

```
B(J,K,I)=...A(I,J,K)
```

Figure B.32: Example in which B can be partitioned in index 2 when A is partitioned in index 3, due to the linear relationship.

If A is partitioned then the assigned values will only exist within a specific range on each processor, therefore if B is mapped and operated upon in a similar way to A, then it would be ideal if B were also partitioned. In Figure B.33 the values of C would be assigned only within the same range as A on each processor, and similarly when using both B and D where their assigned values would only be used at the location where A is assigned. These other variables can use the same partition as A because there is a linear relationship between A and the other variables. For example, the assignment index of A is I, and the usage index of B is also I.

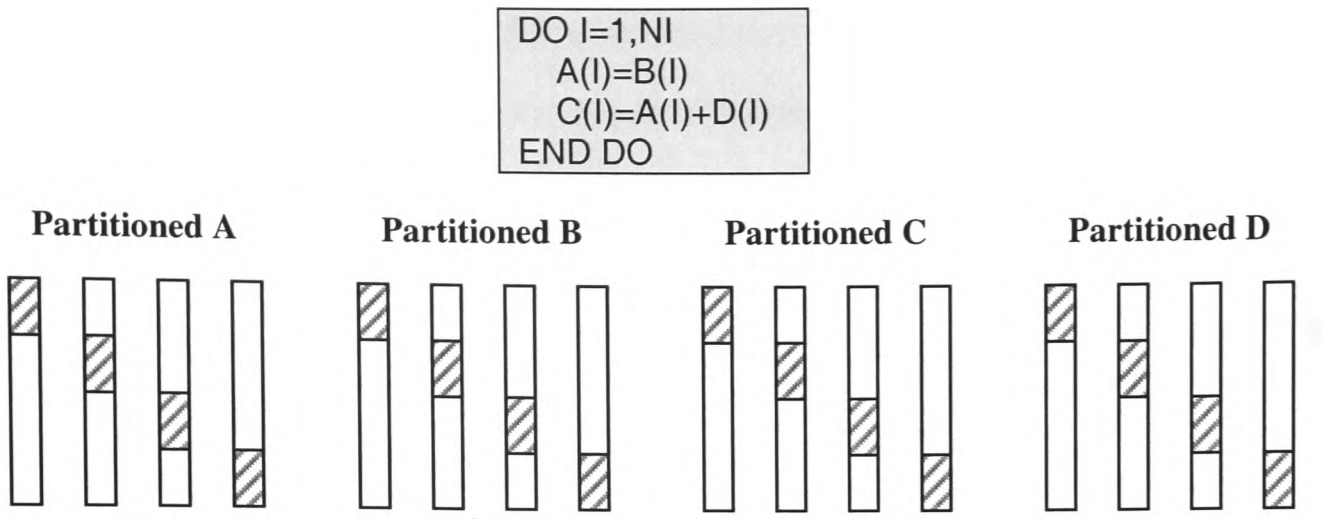


Figure B.33: Demonstrates that arrays B, C, and D can be partitioned since they are aligned with A.

Figure B.34 illustrates that not all arrays can be partitioned, since in this example there is a conflict between the different assignments of E when A is partitioned in the first index. E is said to be 'unpartitioned' since the same partition cannot be used throughout the entire code. E is partitioned in the first index (just like A) in the earlier assignment, but is then partitioned in the second index in the latter assignment (since its I index corresponds with the partitioned I index of A). HPF style redistributions [128, 129] that are very expensive are not used in CAPTools, however unpartitioned arrays do not inhibit parallelisation due to the masking algorithm (Section B.8).

```
DO J=1,NJ
   DO I=1,NI
      E(I,J)=A(I,J)
   END DO
END DO
...
DO J=1,NJ
   DO I=1,NI
      E(J,I)=A(I,J)
   END DO
END DO
```

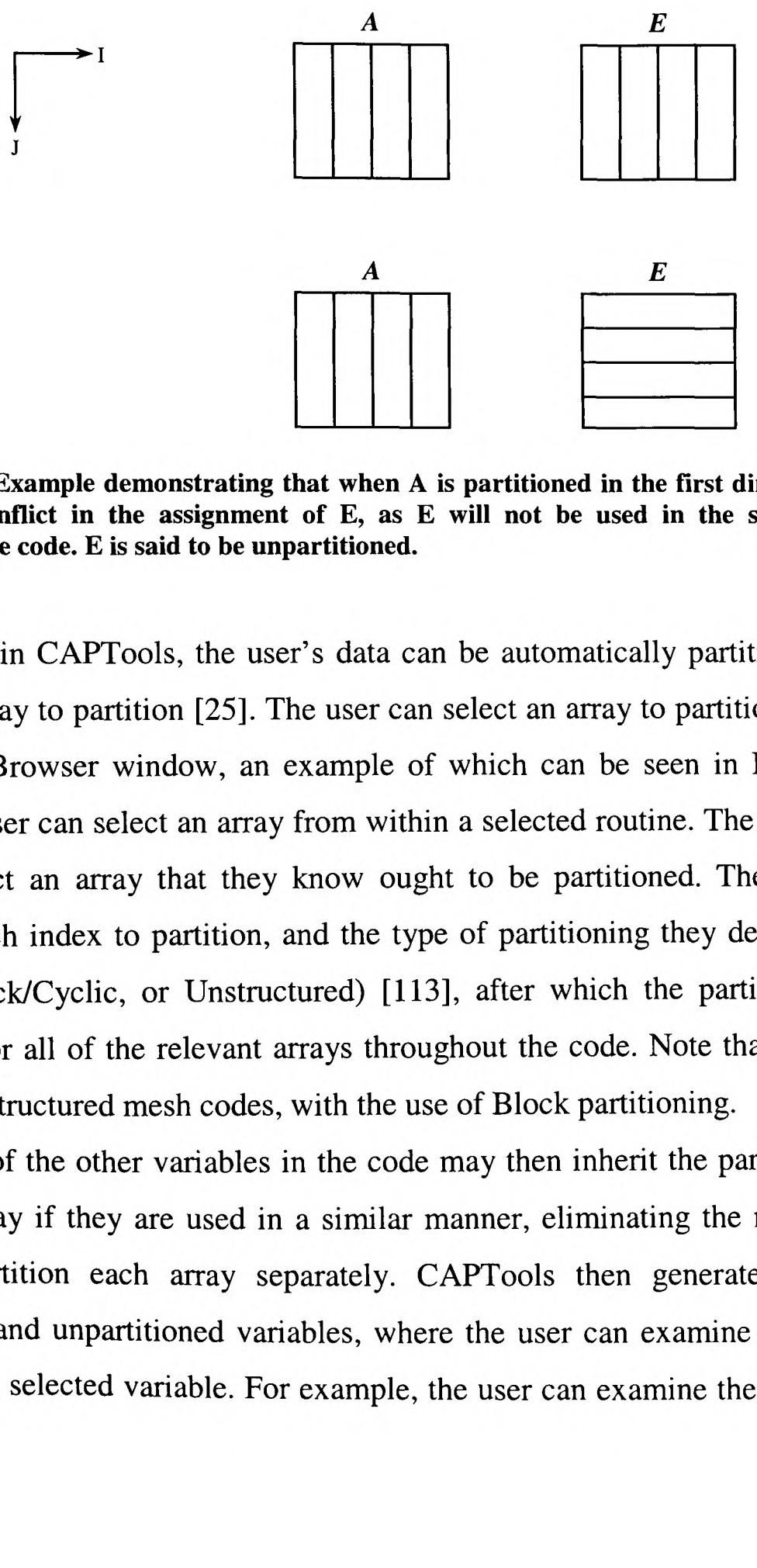


Figure B.34: Example demonstrating that when A is partitioned in the first dimension then there is a conflict in the assignment of E, as E will not be used in the same manner throughout the code. E is said to be unpartitioned.

Within CAPTools, the user's data can be automatically partitioned given an initial array to partition [25]. The user can select an array to partition using the Partitioner Browser window, an example of which can be seen in Figure B.35 where the user can select an array from within a selected routine. The user should ideally select an array that they know ought to be partitioned. They can then choose which index to partition, and the type of partitioning they desire (Block, Cyclic, Block/Cyclic, or Unstructured) [113], after which the partition can be generated for all of the relevant arrays throughout the code. Note that this thesis focuses on structured mesh codes, with the use of Block partitioning.

All of the other variables in the code may then inherit the partition of the selected array if they are used in a similar manner, eliminating the need for the user to partition each array separately. CAPTools then generates a list of partitioned and unpartitioned variables, where the user can examine information relating to a selected variable. For example, the user can examine the reason why

a selected variable has not been partitioned, as well as view information relating to a selected partitioned variable.

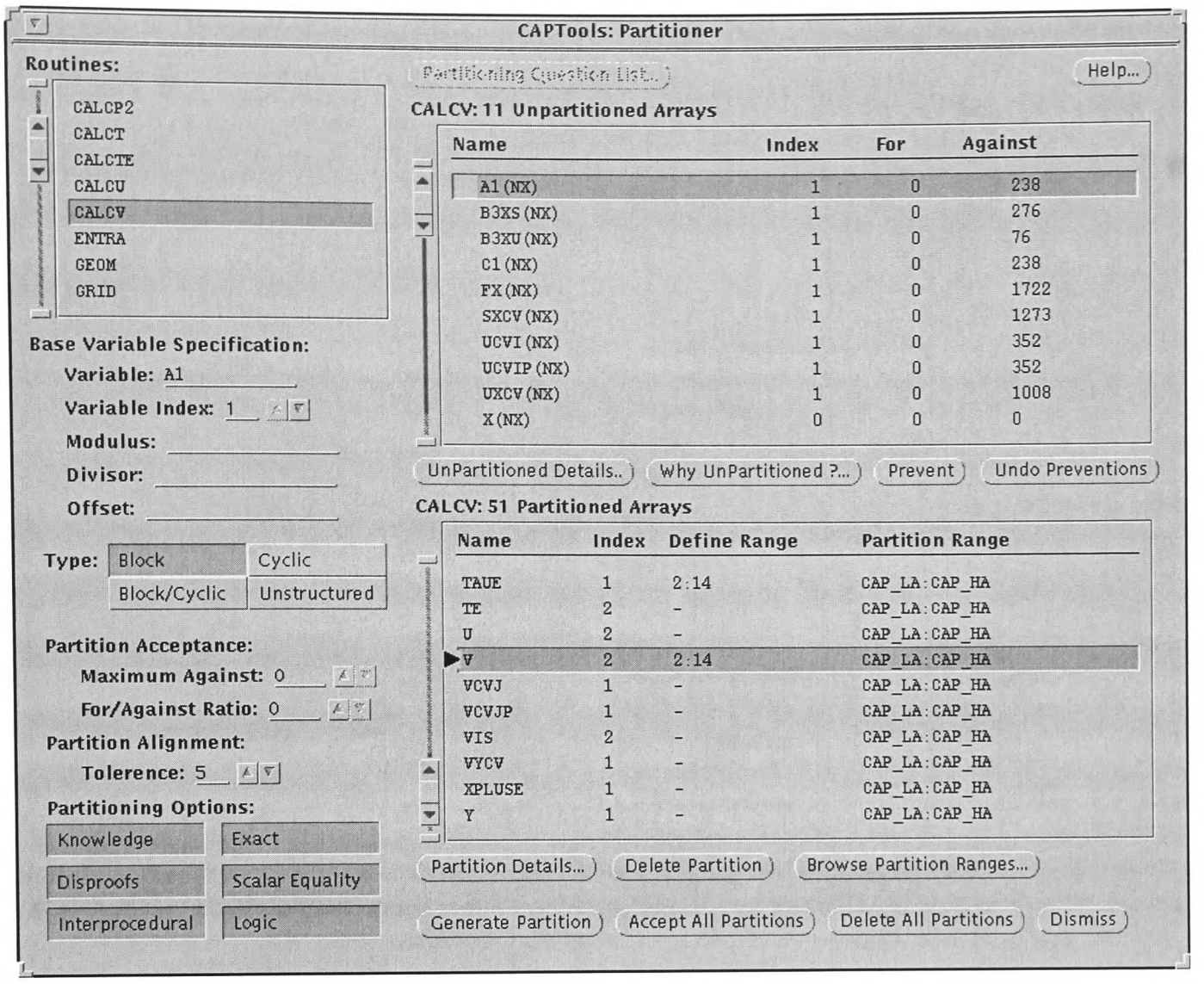
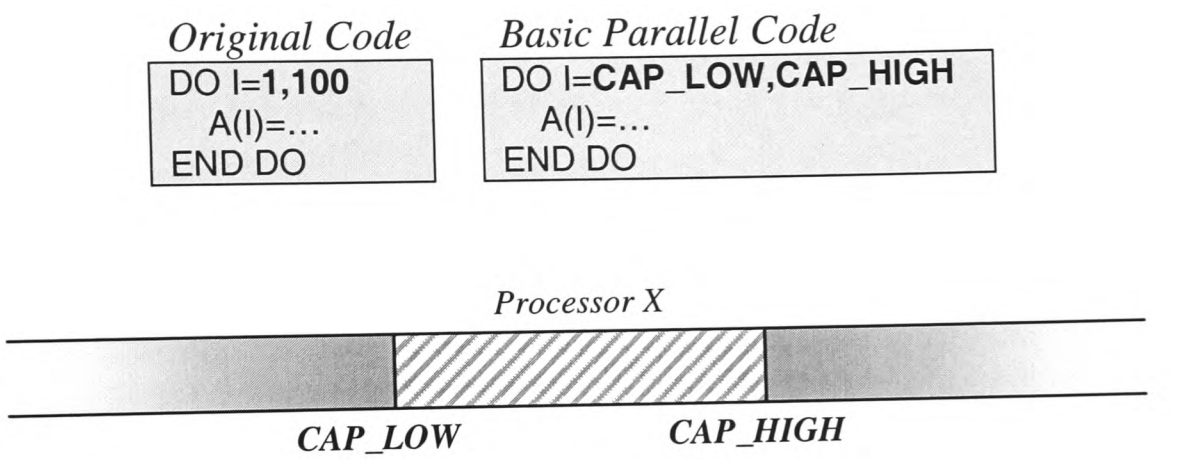


Figure B.35: The Partitioner Browser window within CAPTools.

The user can inspect all of the partitioned arrays in the code by scrolling through the routine list (Figure B.35). The partition list displays the variable name, the partitioned index, the partition range, and the processor partition range limits. Note that the define range are symbolic expressions relating to the range of the array that is assigned and are not necessarily the same as the declaration range, since not all of this declaration range will be operated upon. The processor partition range limit variables are generated by CAPTools and are used throughout the parallel code, as they define the range that each processor 'owns' (operates upon). Each processor shall operate between their lower and higher limits, where their actual values are determined at runtime, as shown in Figure B.36. The number of processors used in the parallel execution is not known until runtime, when it is used to calculate the processor partition range limits along with the symbolic assign ranges of the arrays.

In Figure B.35 the processor partition range limits take the form of CAP_LA and CAP_HA, where the A indicates that the partition is based on the array A. The fact that these limits omit the pass in which the partition was made (Section A.2) indicates that this is the first pass, meaning these limits will change to CAP1_LA and CAP1_HA with any subsequent partitioning. Note that the terms CAP_LOW and CAP_HIGH will often be used throughout this thesis to generally refer to the lower and upper processor partition range limits (ignoring the pass in which they were created).



Processor:	Processor Partition Range Limits	
	CAP_LOW	CAP_HIGH
1	1	25
2	26	50
3	51	75
4	76	100

Figure B.36: Each processor has its own set of processor partition range limits (CAP_LOW and CAP_HIGH) that define its workload, where these limits are determined at runtime. Also shown is an example of the processor partition range limits when the number of processors used is 4.

If the user is not satisfied with any one of the generated partitions then they can either delete or edit the selected partition. A partition can be aligned with another partition within CAPTools where, rather than using numerous partitions throughout the code, it may be more worthwhile aligning the partitions so that just a single, or a few, partitions are used. One example where array partitions can be aligned is shown in Table B.4 where B is aligned with A.

Variable:	Index:	Partition Range:	Processor Partition Range Limits
A	1	1:100	CAP_LA, CAP_HA
B	2	2:99	CAP_LB, CAP_HB

Table B.4: The partition for B can be aligned with the partition for A.

In order to successfully partition all possible arrays in the code, CAPTools must be able to prove without doubt that there is a linear relationship either in a particular statement, between statements, and interprocedurally. The array is unpartitioned if there is no relation to any partitions, or if there are conflicting relationships [25].

B.7.1 Dimension Mapping Between Routines (Modulus And Division)

Different people program in different ways, and so arrays will not always be consistently mapped. Unlike HPF for example where strict regulations must be followed to ensure that each array is consistently mapped (leading to significant porting difficulties) [130], every array need not be defined in the same way throughout the code when using F77. For example, in Figure B.37, the array A is multi-dimensional in the Main routine, but is 1D-mapped (linearised) in Sub1. In this type of situation, CAPTools still needs to be able to extract the partitioned component, for which there are two approaches:

1) using the partition index
2) using the modulus (Mod) and divisor (Div) symbolic expressions

If it is known that the second index of A is partitioned in Main then this allows 'J' to be clearly identified as the partitioned component in this particular routine. For the 1D-mapped reference to array A in Sub1, 'index 2' is meaningless as there is only one index. With CAPTools the partition index for the array A in Main has a value of 2, whereas the partition index for the array A in Sub1 has a non-positive value (indicating that a subset of this index is partitioned). This suggests that the

partitioned component can be obtained using approach 1 in the former case, and only using approach 2 in the latter case.

Any index within an array can be identified either directly using an index number, or indirectly using the Mod and Div values, which is why CAPTools stores both the Mod and Div values for each partitioned component of an array as well as storing the partitioned index. The Mod and Div are used because it would be impossible to identify the partitioned component any other way in the situation when the partitioned index is not known.

The partitioned component of A in Sub1 can be obtained using the Mod and Div values for each individual dimension, shown in Table B.5. The MOD is applied to the 1D expression, after which the DIV is then applied to the remainder term, where the partitioned component can be extracted from the factor term. The 1D expression (I+(J-1)*NI+(K-1)*NI*NJ+(L-2)*NI*NJ*NK) is factorised by the Mod of the second index (NI*NJ), which gives:

Factor=(K-1)+(L-2)*NK
Remainder=I+(J-1)*NI

The remainder term (I+(J-1)*NI) is then factorised by the Div of the second index (NI), which gives:

Factor=(J-1)
Remainder=I

The partitioned component (J) can then be extracted from the factor term, where the 1 is an offset due to linerisation. Symbolic factorisation is applied using the SYMBOLICMOD and SYMBOLICDIV utilities (Table B.3), where legality of the Mod and Div must be proved.

MAIN PROGRAM	SUBROUTINE SUB1(A)
DIM A(NI,NJ,NK,2:NL)	DIM A(*)
...	...
=A(I,J,K,L)	...=A(I+(J-1)*NI+(K-1)*NI*NJ+(L-2)*NI*NJ*NK)
...	...

Figure B.37: Example code in which the array A is multi-dimensional in the Main routine, and is 1D in Sub1.

Index	Div	Mod
1	1	NI
2	NI	NI*NJ
3	NI*NJ	NI*NJ*NK
4	NI*NJ*NK	NI*NJ*NK*(NL-1)

Table B.5: The Mod and Div values for the array A, whose declaration can be seen in the Main routine in Figure B.37.

B.7.2 The Partition Data Structure

CAPTools stores the current partition information in the PARTITION data structure (seen in Figure B.38) where the symbolic name of the partitioned variable, the partitioned index, the processor partition range limits, the Mod and Div values, along with additional information, are stored internally for each partitioned variable in a given routine. The partitioned index can be used to indicate which dimension of the array has been partitioned, where the processor partition range limits define the lower and upper processor partition range limits in which a processor may operate within. Figure B.39 shows the pseudo code for accessing all of the partitioned variables in a given routine and how they can be examined individually by searching the PARTITION data structure. Note that the PARTITION data structure also includes those variables that have been inherited from called routines (Section B.6.7).

```
PARTREC=RECORD
  ROUTINE:PROUTINE;
  SYMBOL:PTABLE;
  MINSYMB,MAXSYMB:PTABLE;
  INDEX:INTEGER;
  MODDIVOFFPTR:PMODDIVOFF;
  NEXT:PPARTITION;
  END;
```

Figure B.38: Sample of the PARTITION data structure record, stored for each routine.

```
PARTITION:=CROUTINE^.PARTITION;
WHILE PARTITION <> NIL DO
  BEGIN
  SYMBOL:=PARTITION^.SYMBOL;
  ...
  PARTITION:=PARTITION^.NEXT;
  END;
```

Figure B.39: Sample code showing how to examine each partitioned variable in a given routine, with the given data structure.

The symbolic name of the variable (PARTITION^.SYMBOL) is simply the symbol table entry of the partitioned variable in the given routine (PARTITION^.ROUTINE). The partitioned index refers to the index of the variable that is linearly related to the partitioned index of the selected array that was chosen by the user (Figure B.35). For instance in Figure B.34, if A is partitioned in the first dimension (index I) then PARTITION^.INDEX:=1, otherwise PARTITION^.INDEX:=2 if A is partitioned in the second index (J). The processor partition range limits (PARTITION^.MINSYMB, and PARTITION^.MAXSYMB) are given for this variable, where they can be different to other limits in the given routine, and they could be different to the limits used in other routines. Figure B.40 shows what would be stored in the PARTITION data structure in the routine SubX if the variable A(200,300) were partitioned in the second dimension, using CAP_LA and CAP_HA, and where B(300,100) were partitioned in the first dimension using the same limits having been aligned with the partition given for A.

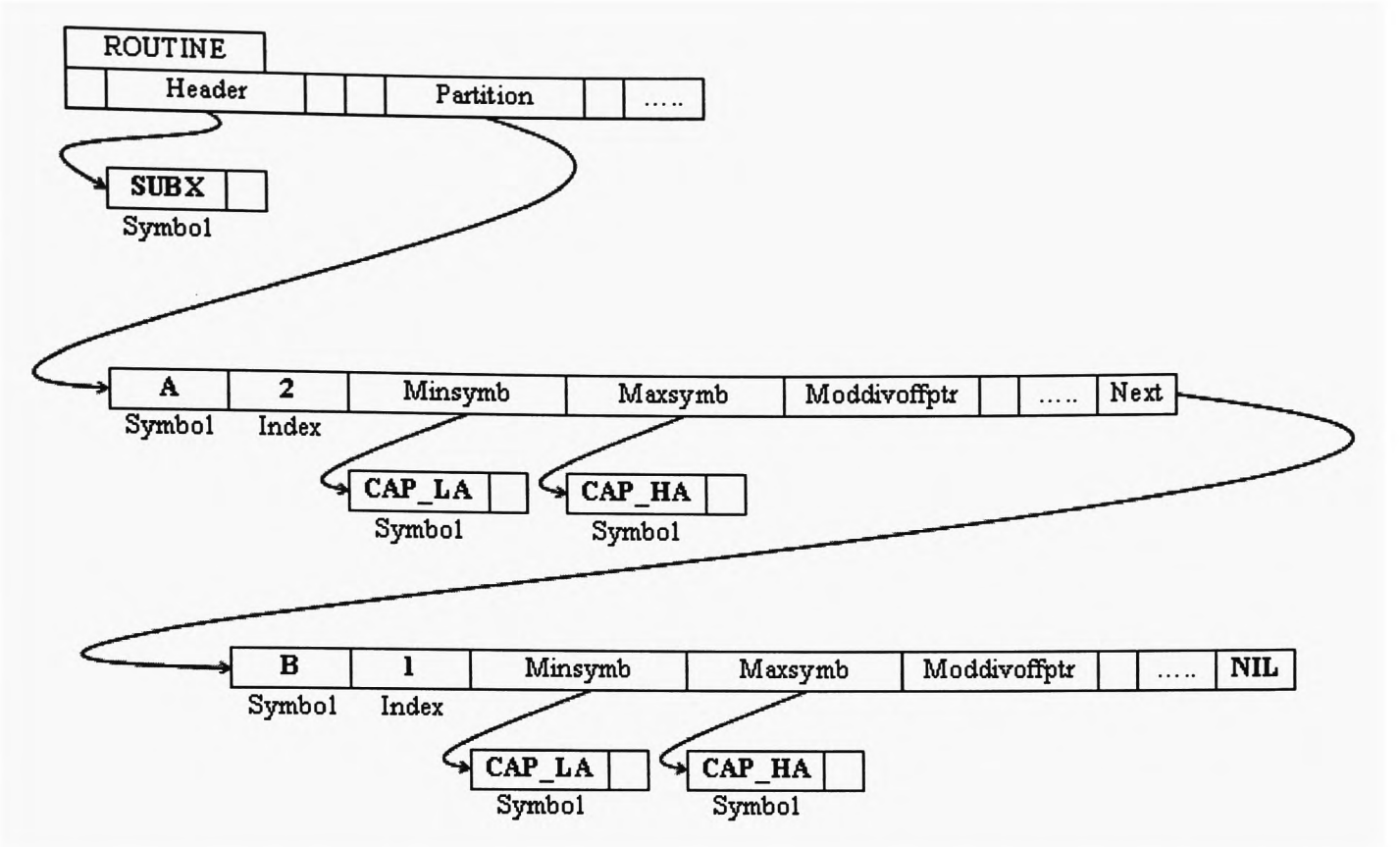


Figure B.40: PARTITION data structure for routine SubX, where both A and B are partitioned.

The Mod and Div values are stored in the MODDIVOFFPTR field of the PARTITION data structure, where the Div value is essentially the stride of the index of interest, and the Mod value is the stride of the next index. The nonloop and constant values are stored within MODDIVOFFPTR for both the Mod and Div, where, continuing with the example given in Figure B.40, the important fields of MODDIVOFFPTR are given in Figure B.41.

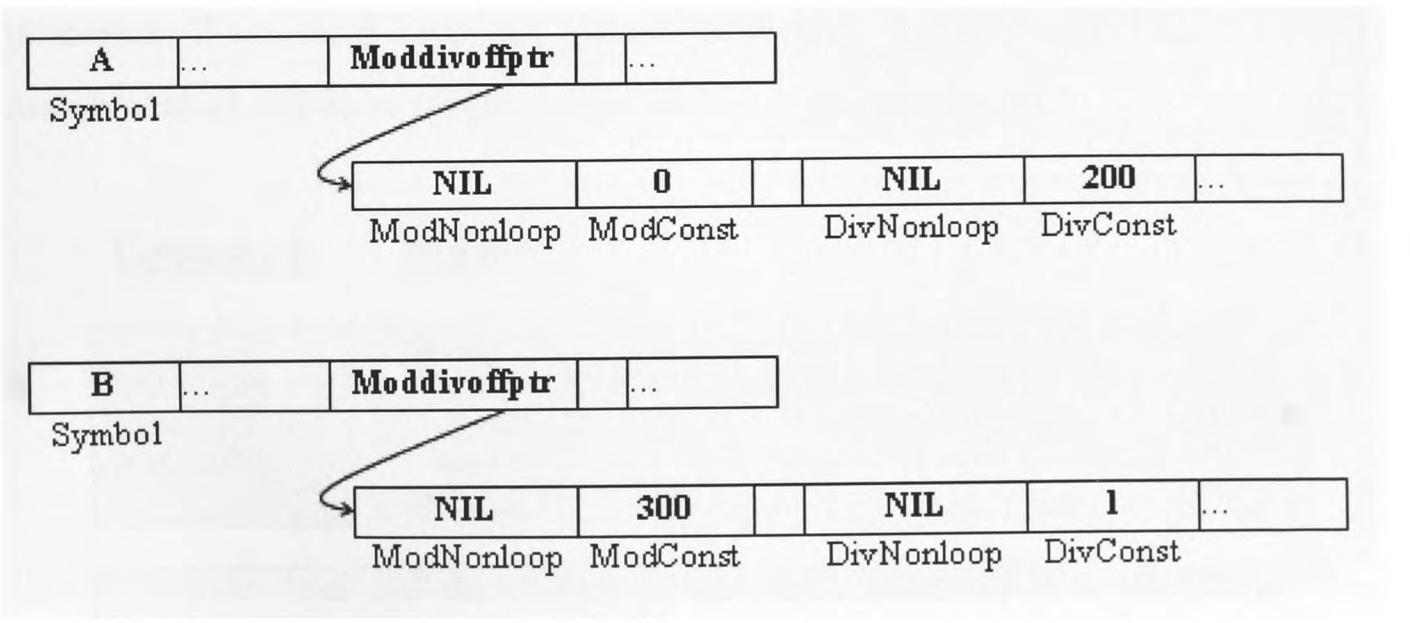


Figure B.41: The MODDIVOFFPTR data structure for A and B in Figure B.40.

B.8 Execution Control Masks

To exploit parallelism, most statements in the code should be made to only execute on a particular processor. A mask simply determines which computations each processor performs, where the more statements that are masked the better, since this means that statements will not be executed unnecessarily on every processor, but only on those that need to execute it. The full set of partitioned arrays is used to enforce the 'assign only allocated data' requirement of the data partition [122]. The execution control masks generally take the form:

```
IF (CAP_LOW<=Expression<=CAP_HIGH) THEN
```

where CAP_LOW and CAP_HIGH are the processor partition range limits for the array that inferred the mask, and the Expression originates from the indices of that array (or a related array), dependent on why the mask was set.

Figure B.42 shows examples in which it would be beneficial to mask the given statements. The first example demonstrates that even the statements within a DO Loop can be masked, such that the calculation is only performed when the partitioned component is between the processor partition range limits of a processor (i.e. the assigned array element is owned). The second example shows the situation in which it is obvious that the statement only needs to be executed on the Processor owning A(NI), since assignments are only made on the owning processor. If the mask were not placed around the boundary calculation, then the assignment of A(NI) would be performed by every processor.

Unmasked:	**Masked:**
DO I=1,NI A(I)=... END DO	DO I=1,NI IF (I.LE.CAP1_HIGH .AND. I.GE.CAP1_LOW) THEN A(I)=... END IF END DO
A(NI)=A(NI)*PI	IF (NI.LE.CAP1_HIGH .AND. NI.GE.CAP1_LOW) THEN A(NI)=A(NI)*PI END IF

Figure B.42: An example of a boundary assignment statement, and array assignment within a loop, which are unmasked and masked.

Ideally, if each of the statements within a block has the same execution control mask then it makes sense to place a single execution control mask around the entire block. The execution control mask could be evaluated just once rather than having to evaluate the mask for each statement, making the parallel code more efficient. However, if the execution control mask of any one of the statements within the block differs then the block cannot inherit the execution control mask. Similarly, when considering the statements within a DO Loop, masked statements can be contained within a single execution control mask, as demonstrated in Figure B.43. CAPTools can transfer the execution control mask onto the DO Loop head itself, such that each processor will essentially operate between their processor partition range limits. For example, for 3 processors, the first processor may operate between 2 and its CAP1_HIGH, the middle processor will operate between their CAP1_LOW and CAP1_HIGH, and the last processor will operate between its CAP1_LOW and NI-1.

```
DO I=2,NI-1
   A(I)=...
   B(I)=...
   C(I)=...
END DO
```

```
DO I=2,NI-1
   IF (I.LE.CAP1_HIGH .AND. I.GE.CAP1_LOW) A(I)=...
   IF (I.LE.CAP1_HIGH .AND. I.GE.CAP1_LOW) B(I)=...
   IF (I.LE.CAP1_HIGH .AND. I.GE.CAP1_LOW) C(I)=...
END DO
```

```
DO I=2,NI-1
   IF (I.LE.CAP1_HIGH .AND.
         I.GE.CAP1_LOW) THEN
      A(I)=...
      B(I)=...
      C(I)=...
   END IF
END DO
```

```
DO I=MAX(2,CAP1_LOW),
         MIN(NI-1,CAP1_HIGH)
   A(I)=...
   B(I)=...
   C(I)=...
END DO
```

Figure B.43: An example in which the execution control masks of the individual statements within a block can first be transferred to the block itself, and then to the surrounding loop head. In each case the execution control masks are the same for all of the statements in the block, and are the same for all of the blocks within the DO Loop.

Every executable statement within the code can either be masked or unmasked, where it would be ideal if execution control masks could be combined when possible. Another instance in which the execution control masks can be combined is when all of the statements within a subroutine have the same mask. When this happens, it is possible to remove the masks from the statements inside the subroutine and place a single execution control mask around the calls to that particular subroutine (Figure B.44).

```
IF (I.LE.CAP1_HIGH .AND. I.GE.CAP1_LOW)
   CALL SUB1
END IF
```

```
IF (I.LE.CAP1_HIGH .AND. I.GE.CAP1_LOW)→
IF (I.LE.CAP1_HIGH .AND. I.GE.CAP1_LOW)→
IF (I.LE.CAP1_HIGH .AND. I.GE.CAP1_LOW)→
IF (I.LE.CAP1_HIGH .AND. I.GE.CAP1_LOW)→
```

```
....
SUBROUTINE SUB1
S1
S2
S3
S4
END
```

Figure B.44: Example in which the execution control mask has been placed around the call to Sub1 since all of the statements in Sub1 have the same execution control masks.

CAPTools examines each statement within the code, deciding whether or not the statement should be masked or unmasked. Partitioned data is used as a start point for comprehensively adding execution control masks where the majority of execution control is inferred from other statements rather than strictly enforced through partitioned data.

Parallel execution of a code on a DMS is only possible for masked statements, since unmasked statements will definitely be executed on every single processor. Several rules can therefore be used to try to achieve the maximum coverage of execution control masks [25] (Figure B.45), aiming to maximise parallelism and also to minimise the frequency and volume of communication (Section B.9).

Rule 1. *Assignment of partitioned data (owner computes assignment). This is due to the 'assign only allocated data' requirement. For example, A has been partitioned in index 1, and so this assignment should be made only between the processor partition range limits on every processor.*

Rule 2. *Usage of partitioned data (owner computes usage). The statement is only executed on the processor that owns the up-to-date values of the used partitioned array.*

Rule 3. *Assignment of data that is only used by masked statements. This imposes control on a statement to only execute on the processor(s) where the assigned data is required. The statement that assigns X has inherited its execution masks from the two statements that use the variable X. Masks of this type are unsafe if all of the statements using the assigned data are not masked.*

Rule 4. *Usage of data that is only assigned in masked statements. This imposes control on a statement to only execute on a processor where the used data already resides. The statement that uses X inherits the execution mask from the first statement due to the usage of the variable X.*

Rule 1	*IF (CAP_LOW<=I<=CAP_HIGH) A(I,J)=…*
Rule 2	*IF (CAP_LOW<=I<=CAP_HIGH) …=A(I,J)…*
Rule 3	*IF (CAP_LOW<=1<=CAP_HIGH or* *CAP_LOW<=N<=CAP_HIGH) X=…* … IF (CAP_LOW<=1<=CAP_HIGH) A(1,J)=…X… IF (CAP_LOW<=N<=CAP_HIGH) A(N,J)=…X…
Rule 4	IF (CAP_LOW<=N<=CAP_HIGH) X= *IF (CAP_LOW<=N<=CAP_HIGH) …=…X…*

Figure B.45: Rules and examples aiming to try and ensure maximum coverage of execution control masks.

The union of all the masks of the statements within a routine can be inherited by the routine call. This implies that if any of the statements within that routine are unmasked (or is not a subset of the union of all other masks) then the call to that routine cannot be masked. Both partitioned and unpartitioned array accesses are masked, so for example, in Figure B.34 a mask (using Rule 2) can be placed around the assignment statements of the variable E, such that E is treated as if it is partitioned in each case. Note that E would not be put in the partition list

(Figure B.35) because E is only implicitly partitioned during the masking phase of the parallelisation process.

Another instance in which execution control masks are clearly needed is when handling I/O, as one processor (usually Processor 1) should ideally do this. In the absence of parallel I/O for example, if every processor were to read in the dimensions of the structured mesh problem then as well as being inefficient this would become very frustrating for the user, especially if 100's of processor were used. The master usually deals with any I/O, where either data is read in and sent out to all the other processors if required, or data is received from the other processors and written out.

The user can generate the execution control masks for their code by selecting the Generate Masks option in the Code Generator window (Figure B.46). CAPTools will then produce a list of all the masked and unmasked statements in every routine, which the user can examine, as demonstrated in Figure B.47.

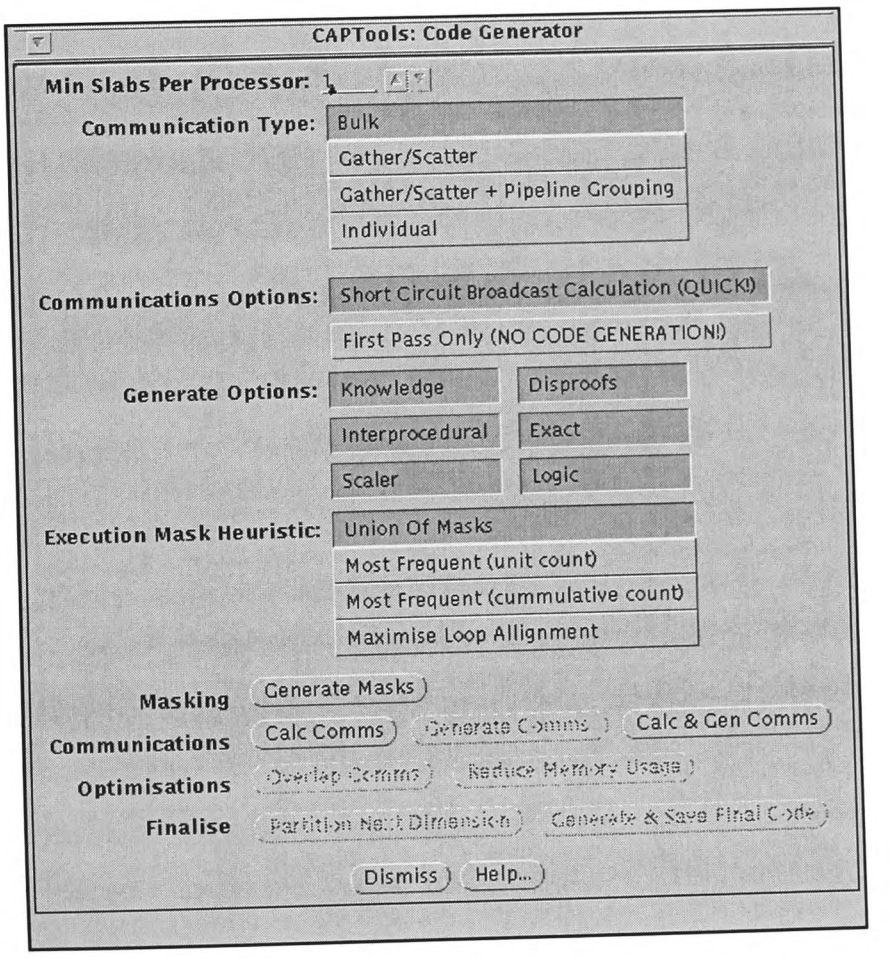


Figure B.46: Code Generator window in CAPTools.

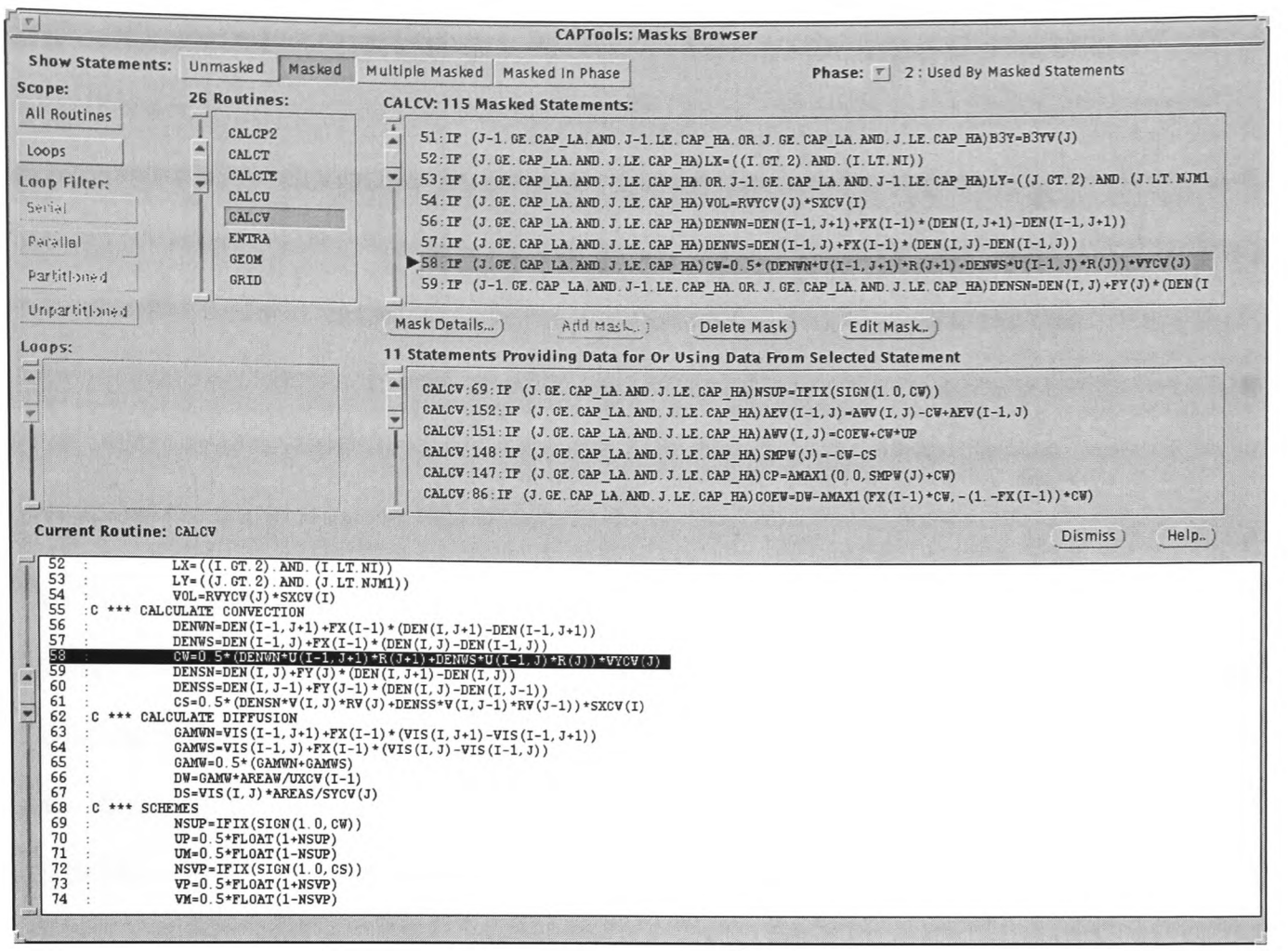


Figure B.47: The Mask Browser window enables viewing of all masked and unmasked statements generated in the current pass.

CAPTools stores the masking information for every command in the MASK field of that COMMAND, where its data structure can be seen in Figure B.48. The actual execution control mask can be obtained from LINK pointing to a symbolic expression (Section B.6.10), and the PARTITION field of the relevant array. The MASKTYPE and PHASE are respectively used to indicate the type of mask (e.g. BLOCK etc) and the rule (Figure B.45) in the masking algorithm by which this mask was created. The APPLYMODDIV field is used to indicate whether or not the Mod and Div expressions need to be applied at runtime to extract the mask component (i.e. where symbolic extraction has failed), and NEXT points to the next mask on the command.

```
MASK=RECORD
   LINK:PLINKER;
   PARTITION:PPARTITION;
   MASKTYPE,PHASE:INTEGER;
   APPLYMODDIV:BOOLEAN;
   NEXT:PMASK;
   END;
```

Figure B.48: Part of the MASK data structure for a command.

B.9 Communications

The previous stages in the parallelisation process involve ensuring each processor computes on their own workload defined by their processor partition range limits. In parallel processing, each processor must operate on current and up-to-date data, where a communication is needed when one processor assigns (owns) that data, and another processor needs to use that data. For instance, this is typically the case in calculations involving data that is in the halo region. If this data is not transferred onto the using processor before it is used, then this will lead to an incorrect solution of the problem. When manually parallelising a code the user will determine which data needs to be exchanged between processors, including how much data, and to whom this data needs to be communicated. CAPTools performs this same task automatically, identifying what needs to be communicated, and where to place these communication calls [25, 28].

B.9.1 The Calculation And Generation Of Communications

Several steps are used to calculate and generate the communications, namely calculating the communication request control sets, migrating these requests 'up' through the code to execute as early on as possible, merging the requests, and finally generating the actual communications. Before discussing each of these steps in more detail, consider first the example shown in Figure B.49. The value of T is assigned between the processor partition range limits of each processor in statement S2, and similarly for the assignment of U in statement S3 where certain values of the assigned data are required on other processors in statements S8, S12, and S16. For example, when I=CAP1_LOW in statement S8, the processor assigning the value of R(I) will need to use a value of T that was assigned on a neighbouring processor (in the lower direction), since T(CAP1_LOW-1) is not owned by the assigning processor. In this example, when I is anything other than CAP1_LOW, the value of T(I-1) will have been calculated on the assigning processor and so a communication is only required to update the lower halo region when I=CAP1_LOW. Similarly, when I=CAP1_LOW in statement S12, the value

of T(CAP1_LOW-2) will have been assigned on a neighbouring processor, and likewise for the value of U(CAP1_LOW-1), suggesting the need to communicate this data. Additionally the value of T(CAP1_LOW-1) will also be needed when I=CAP1_LOW+1 in statement S12. Finally, in statement S16, although the assignment is only made on the processor owning T(NI), it may be possible that the value of T(NI-1) could have been assigned on a neighbouring processor, suggesting the need to communicate this data to the using processor (the processor making this assignment). If the minimum number of slabs (a column, row, plane, etc, of cells) on each processor is set to 1, then it is possible that T(NI-1) could be on a neighbouring processor and likewise T(CAP1_LOW-2) could be on a neighbour's neighbour. In the latter case, the value of T(CAP1_LOW-1) would have to be updated before the value of T(CAP1_LOW-2), as the value of T(CAP1_LOW-2) could be the value of the neighbour's T(CAP1_LOW-1), as illustrated in Figure B.50a. If, however, the minimum number of slabs on each processor is set to 2, then the value of T(CAP1_LOW-2) will definitely be found on the neighbouring processor, also shown in Figure B.50b.

The user can specify the minimum number of slabs (MIN_SLABS) in the Code Generator window (Figure B.46) where the default value is set to 1. The value of MIN_SLABS can be used to symbolically express the fact that the value of CAP1_HIGH is always greater than the value of CAP1_LOW, which is otherwise unknown to CAPTools. If MIN_SLABS is set to 1, then each processor will own at least one slab of cells, where CAP_HIGH $\geq$ CAP_LOW. If MIN_SLABS is set to 2, then each processor will own at least two slabs of cells, where CAP_HIGH $\geq$ CAP_LOW+1.

```
S1    DO I=MAX(1,CAP1_LOW),MIN(NI,CAP1_HIGH)
S2       T(I)=...
S3       U(I)=...
S4    END DO
S5    B=...
S6    IF (C1) THEN
S7       DO I=MAX(2,CAP1_LOW),MIN(NI-1,CAP1_HIGH)
S8          R(I)=T(I)+T(I-1)
S9       END DO
S10   ELSE
S11      DO I=MAX(3,CAP1_LOW),MIN(NI,CAP1_HIGH)
S12         V(I)=V(I)-T(I-2)+U(I-1)
S13      END DO
S14   END IF
S15   IF (NI.LE.CAP1_HIGH .AND. NI.GE.CAP1_LOW) THEN
S16      T(NI)=T(NI-1)
S17   END IF
```

Figure B.49: Example demonstrating that there are several usages of the assigned data, each requiring data on a neighbouring processor.

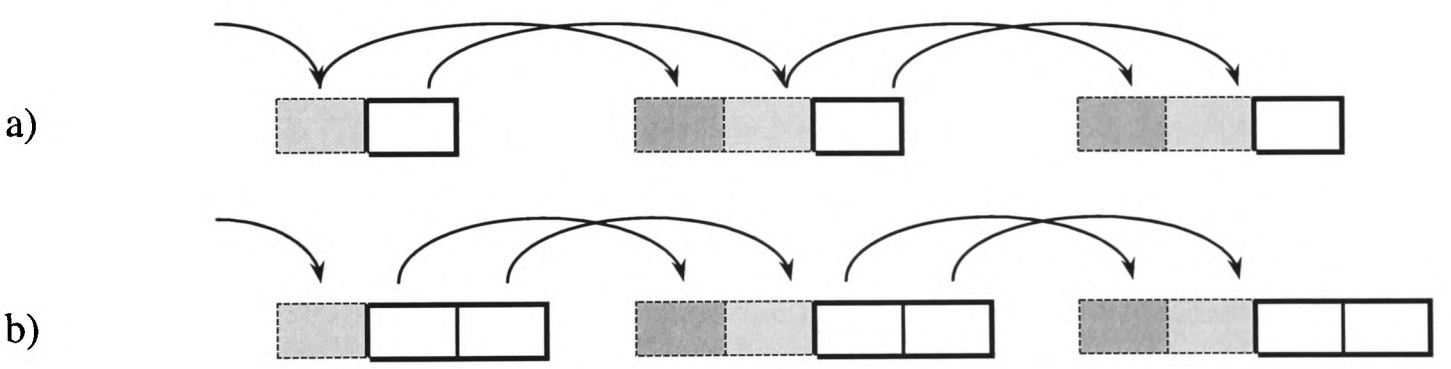


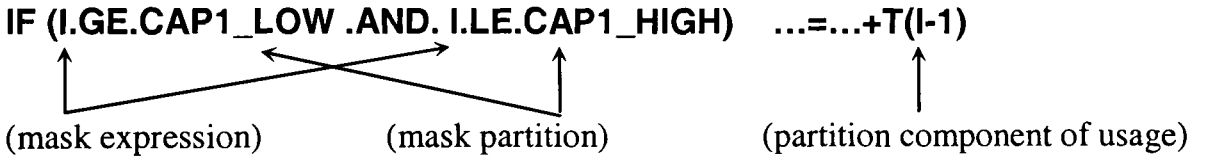
Figure B.50: Example illustrating the need to communicate T(CAP1_LOW-1) before communicating T(CAP1_LOW-2) when a) MIN_SLAB=1, where the former is represented by the lightly shaded region, and the latter is represented by the heavily shaded region; and b) when MIN_SLAB=2, both T(CAP1_LOW-1) and T(CAP1_LOW-2) are both on a neighbouring processor.

B.9.1.1 Calculation Of Communication Requests

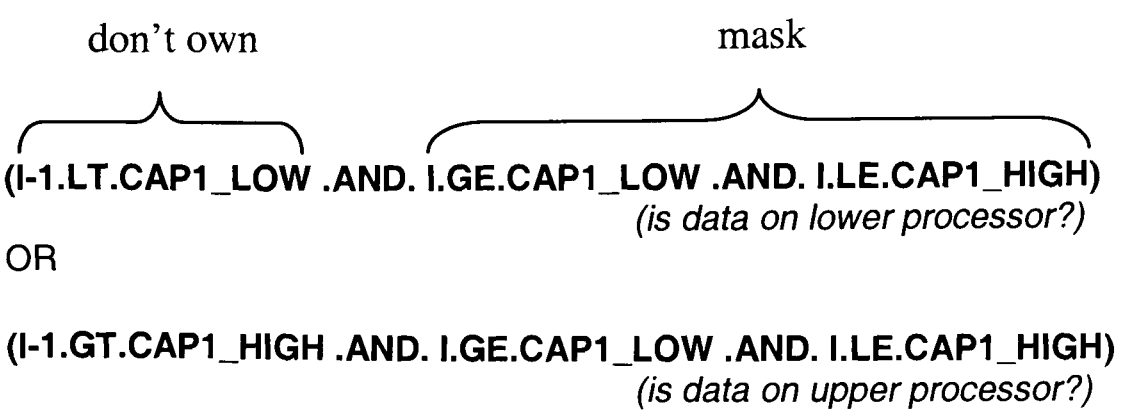
CAPTools identifies a 'request' for data when a processor needs to use data that has been assigned on another processor. The usage of the data is examined, where a communication is required if the usage is not on the same processor as the assignment, i.e. the basic concept of "where data is needed versus where data is owned", which can be expressed using a control set.

If the data is a partitioned array, then CAPTools compares the mask on the usage statement and the partition component of the used array. Since the processor partition range limits are only assigned definite values at runtime, the calculation of communication requirements must therefore rely on equality (or other relationships) between these variables using the symbolic inequality disproof algorithm and related algorithms (Section B.6.10.3). Note that a control set is calculated for each usage in a statement. For example, if a statement contained the expression =W(I-1)+W(I+1) then one control set would be calculated for the usage of W(I-1), where a separate control set would be calculated for the usage of W(I+1).

In Figure B.49 for example, the communication control set (for each communication direction) for the value of T(I-1) in statement S8 can be obtained as follows where the usage partition of T can be found in the partition list for the routine containing this statement:



The communication control set of T(I-1) in statement S8 is based on the partition component (I-1) of the used array T, and the mask of the usage statement. One control set is used to test for communication requests in the lower direction (CAP_LEFT, CAP_UP, etc), and the other is used to test for communication requests in the upper direction (CAP_RIGHT, CAP_DOWN, etc), as data could be required in either or both directions. The set is the intersection of the "don't own used data" and the mask control:



Normalisation of the communication control set is obtained by substituting a dummy variable IC into the control set in place of the partition component (i.e. IC=I-1 in this example):

```
(IC.LT.CAP1_LOW .AND.
        IC+1.GE.CAP1_LOW. AND. IC+1.LE.CAP1_HIGH)
```

OR

```
(IC.GT.CAP1_HIGH .AND.
        IC+1.GE.CAP1_LOW .AND. IC+1.LE.CAP1_HIGH)
```

Figure B.51 shows the graphical representation (on one processor) of the control sets for T(I-1) in statement S8, where a communication request is made if all of the conditions in either of the control sets are true. For each condition in the control set, the lightly shaded region indicates when that condition is true, where the heavily shaded region indicates when all of the conditions of the particular control set are true.

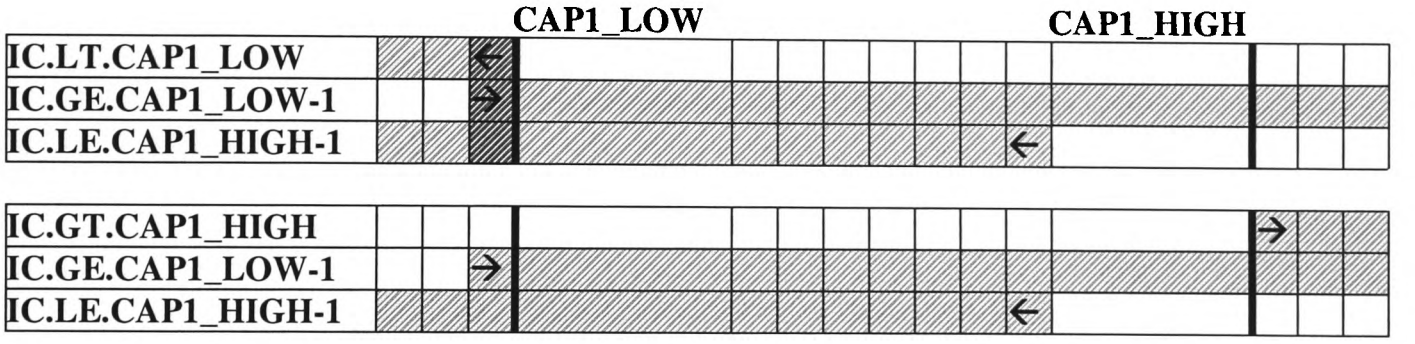


Figure B.51: Graphical representation of the control sets for T(I-1) in statement S8 on a single processor, where the lightly shaded region indicates when a condition is true, and the heavily shaded region indicates when all of the conditions of a particular control set are true.

In the first control set, relating to data on a lower processor, the inference engine can ignore the last condition (IC+1.LE.CAP1_HIGH) using the knowledge that CAP1_LOW is always less than or equal to CAP1_HIGH. If IC is less than CAP1_LOW (the first condition) then IC will also be less than or equal to CAP1_HIGH-1. The second control set, relating to data on an upper processor, can be removed since the first and last conditions are contradictory. The normalised control sets can therefore be simplified to:

```
(IC.LT.CAP1_LOW .AND. IC.GE.CAP1_LOW-1)
```

where it can be seen that a communication is required when IC=CAP1_LOW-1 (the region in which the conditions overlap). The communication control set for U(I-1) in statement S12 is calculated to be the same (IC=CAP1_LOW-1), as its control set is evaluated to be the same as that for T(I-1).

The normalised communication control set for T(I-2) in statement S12 (represented graphically in Figure B.52) can be found in a similar manner, where IC=I-2:

```
(IC.LT.CAP1_LOW .AND.
        IC+2.GE.CAP1_LOW .AND. IC+2.LE.CAP1_HIGH)

OR

(IC.GT.CAP1_HIGH .AND.
        IC+2.GE.CAP1_LOW .AND. IC+2.LE.CAP1_HIGH)
```

In this instance, a communication is required when IC=CAP1_LOW-1 and when IC=CAP1_LOW-2 (i.e. when CAP1_LOW-2 ≤ IC ≤ CAP1_LOW-1).

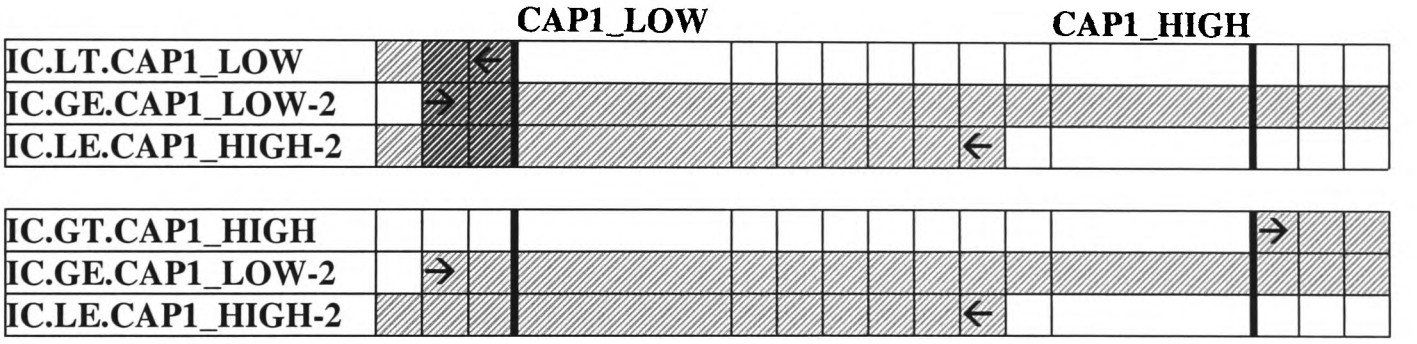


Figure B.52: Graphical representation of the control sets for T(I-2) in statement S12 on a single processor, where the lightly shaded region indicates when a condition is true, and the heavily shaded region indicates when all of the conditions of a particular control set are true.

The communication control set for T(NI-1) in statement S16 is again formed in the same manner as above:

```
(NI-1.LT.CAP1_LOW .AND.
        NI.GE.CAP1_LOW .AND. NI.LE.CAP1_HIGH)

OR

(NI-1.GT.CAP1_HIGH .AND.
        NI.GE.CAP1_LOW .AND. NI.LE.CAP1_HIGH)
```

where the normalised control set is slightly different, in the sense that it includes IC=NI-1:

```
(IC.LT.CAP1_LOW .AND.
        IC+1.GE.CAP1_LOW .AND. IC+1.LE.CAP1_HIGH .AND. IC=NI-1)

OR

(IC.GT.CAP1_HIGH .AND.
        IC+1.GE.CAP1_LOW .AND. IC+1.LE.CAP1_HIGH .AND. IC=NI-1)
```

In this case, a communication is required when IC=CAP1_LOW-1 and IC=NI-1 (where NI-1 will be on a neighbouring processor in certain situations when MIN_SLAB=1, see Figure B.57).

B.9.1.2 The Communication Of Implicitly Partitioned Data

Unpartitioned data can be handled in the same way as partitioned data if both the assignment and usage statement of the data are masked (in the same pass), as illustrated in Figure B.53, where the assignments can be identified by examining the dependencies of the usage. In this example, each processor assigns the value of V(J+1) with J between their processor partition range limits (i.e. V will be assigned between CAP1_LOW+1 and CAP1_HIGH+1), as illustrated graphically in Figure B.54. When using the values of V(I) with I between the processor partition range limits (as seen in the usage statement in Figure B.53), the value of V(CAP1_LOW) will be unknown, since it was assigned on a neighbouring processor. Examining the key dependence of the usage and assignment statements, a linear relationship can be proved, where the communication request control sets can be normalised [25].

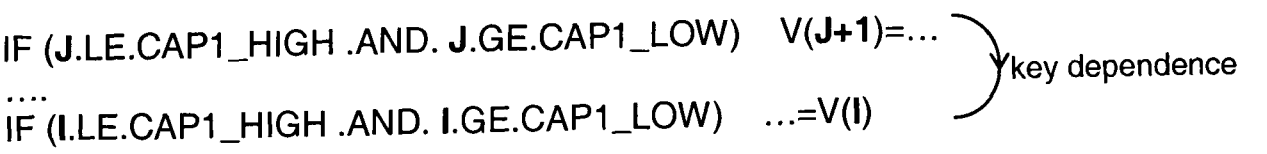


Figure B.53: Example illustrating that data is needed from a neighbouring processor even when the data is unpartitioned using True dependencies.

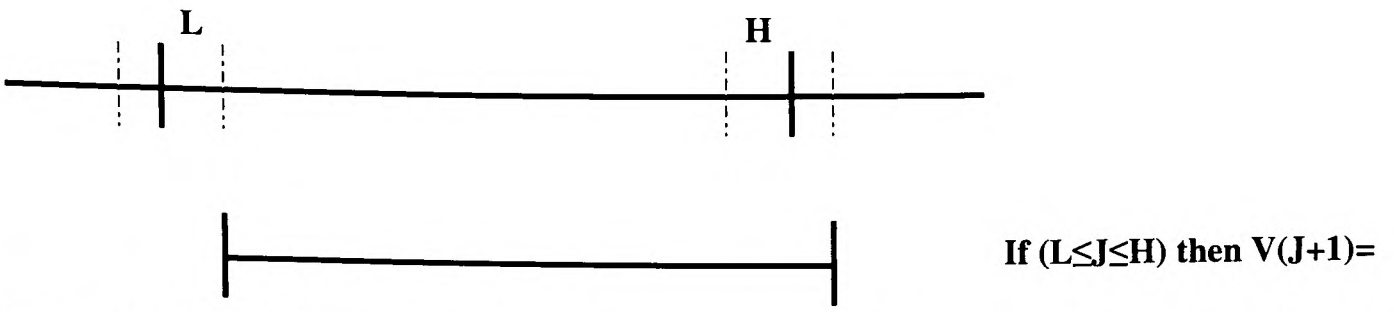


Figure B.54: Graphical illustration of the assignment of the unpartitioned data V in the example shown in Figure B.53. Each processor assigns values of V between their CAP1_LOW+1 and CAP1_HIGH+1 (in which L represents CAP1_LOW and H represents CAP1_HIGH), implying the value of V(CAP1_LOW) is assigned on a neighbouring processor.

B.9.1.3 Conflict Broadcasts

Data will need to be broadcast (Section A.3.3.5) when it is unknown where the correct data resides since it is possible that the data may be assigned on several processors with differing assignment masks (Figure B.55). In such circumstances the data will need to be broadcast to every processor after the assignment (conservative solution) as it is difficult to ascertain which processor made the assignment. If this data is not broadcast then it is possible that the wrong value will be used.

Which value of X will be used in subsequent code?

```
IF (5.LE.CAP1_HIGH .AND. 5.GE.CAP1_LOW) THEN
   X=90
END IF
Broadcast X from processor who made assignment to the
Left using Send and Receive statements
Broadcast X from processor who made assignment to the
Right using Send and Receive statements

IF (9.LE.CAP1_HIGH .AND. 9.GE.CAP1_LOW) THEN
   X=50
END IF
Broadcast X from processor who made assignment to the
Left using Send and Receive statements
Broadcast X from processor who made assignment to the
Right using Send and Receive statements
```

Figure B.55: Example illustrating conflict broadcasts.

B.9.1.4 Migration Of Communication Requests

The calculated communication requests need to be migrated up through the code as far as possible to reduce the frequency of executions. For example in Figure B.49, rather than communicate the value of T(CAP1_LOW-1), used in S8, inside the S7 loop, the communication could be executed just once if placed before the loop. The same is true for the communication of T(CAP1_LOW-1), T(CAP1_LOW-2), and U(CAP1_LOW-1) that are used in S12 inside the S11 loop. CAPTools aims to reduce the communication latency by migrating the requests out from the loop and up the code as far as possible.

For instance, considering the latter example involving the communication control sets for statement S12, these communications can be executed in several ways, as shown in Figure B.56 (where the requested data is communicated before its usage). As demonstrated in Figure B.56a, the three communications (satisfying each request of statement S12) can be placed inside the loop where they are executed every iteration. The requested data is not needed for every value of I, implying that the data would be communicated unnecessarily every iteration, leading to high communication latencies. Although the communication latencies are reduced in Figure B.56b, in which the communications are only executed for particular values of I, the test on these values of I incur their own penalty. It is possible to migrate the requests out from this loop since the requested data is not assigned in this loop. In Figure B.56c, the requested data is communicated just once before the actual DO Loop, where no tests are needed on the value of I.

a)	DO I=... comm(T(CAP1_LOW-1),...) comm(T(CAP1_LOW-2),...) comm(U(CAP1_LOW-1),...) ...=T(I-2)+U(I-1) END DO
b)	DO I=... IF(I.EQ.CAP1_LOW+1)THEN comm(T(CAP1_LOW-1),...) END IF IF(I.EQ.CAP1_LOW)THEN comm(T(CAP1_LOW-2),...) END IF IF(I.EQ.CAP1_LOW)THEN comm(U(CAP1_LOW-1),...) END IF ...=T(I-2)+U(I-1) END DO
c)	comm(T(CAP1_LOW-1),...) comm(T(CAP1_LOW-2),...) comm(U(CAP1_LOW-1),...) DO I=... ...=T(I-2)+U(I-1) END DO

Figure B.56: Example illustrating the possible locations at which to satisfy the communication request control sets of the S11 loop in Figure B.49, where the data can a) be updated every iteration; b) be updated only for specific iterations; or c) be updated just once before the loop.

CAPTools tries to migrate the request as far up the code as possible, where the communications are executed preferably just after the assignment of the used data, and it is hoped the requests can be merged (Section B.9.1.5). Figure B.56 demonstrated that the communication requests for the statement S12 can be migrated out of the S11 loop, where the communications will be executed before the usage of the requested data, however, it is possible to migrate these requests even further. CAPTools uses the predominator tree (Section B.4.1) to migrate the requests to a location where they will definitely be executed, where any barriers, such as the assignment statement of the used data, or a loop containing such an assignment, will halt further migration of the request. In Figure B.49 the requests for the statement S12 can therefore be migrated up as far as S5 (before S5 and S6), where they will still be executed before the requested data is used. The communication requests cannot be migrated further up the code due to the barrier caused by the S1 loop that contains the assignment of the used data. In fact, all of the communication requests for the example in Figure B.49 can be migrated to execute before statement S5. For example, the request for statement S16 is first

migrated from before S16 to before S15, after which it is migrated to before S6, and then to before S5. Use of the predominator tree when migrating the communication requests guarantees that the communications will always be executed before the usage of that data.

Migrating a request 'up' the code using the CFG (Section B.4) in a particular routine tries to ensure earlier execution of the communication in that routine. A request can be migrated to the top of a routine, where it is then possible to migrate the request even further, using the call graph (Section B.3), into the calling routine(s).

B.9.1.5 Merging Communication Requests

Each assigned variable may have many usage statements, meaning each of these usage statements could potentially make a request for the data to be communicated. The communication latency would be extremely high, as well as unnecessary, if a communication were generated for each request, as the same data may be communicated several times. The requested data should ideally be communicated just once, such that all of the usage statements will have access to the communicated data. Communication requests are merged together to reduce repeated data transfer, and to reduce the startup latency cost. Several requests for a particular variable can only be merged together if they (the requests) are at the same location in the code, emphasising the need to migrate the requests up through the code (Section B.9.1.4). Migration of the communication requests enables the requests to be merged, as it is unlikely that most communications will naturally occur at the same location. Interprocedural migration is beneficial since it allows requests from different routines (or functions) to be merged.

In Figure B.49 for example, several requests were made for T(CAP1_LOW-1), all of which were migrated to execute before statement S5. To avoid transferring T(CAP1_LOW-1) several times, the requests for this data can be merged so that the data is only communicated once. The control sets of the communication requests for a particular variable are compared, where any requests that are subsets of other requests can be merged into a single request. For

example, (IC=CAP1_LOW-1 and IC=NI-1) can be merged with (IC=CAP1_LOW-1) as the former control set is a subset of the latter control set. Whenever (IC=CAP1_LOW-1 and IC=NI-1) is true, then (IC=CAP1_LOW-1) is also true, so it can be merged into (IC=CAP1_LOW-1), which is represented graphically in Figure B.57 illustrating the situation in which most processors (including the last processor) own only one cell. Additionally, the control set (IC=CAP1_LOW-1) is a subset of (CAP1_LOW-2 ≤ IC ≤ CAP1_LOW-1), leaving just two communications requests for this small example, as shown in Figure B.58.

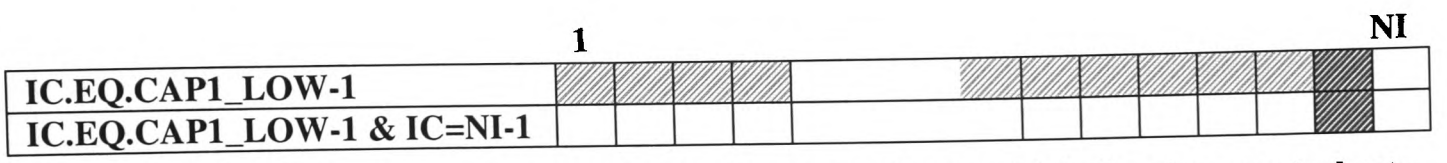


Figure B.57: Graphical representation indicating the region in which the given control sets are true (lightly shaded), and the region in which both control sets are true (heavily shaded), where most processors own just one cell (MIN_SLAB=1). In this example, it is possible to merge the control sets, since (IC=CAP1_LOW-1 and IC=NI-1) is a subset of (IC=CAP1_LOW-1).

```
S1   DO I=MAX(1,CAP1_LOW),MIN(NI,CAP1_HIGH)
S2     T(I)=...
S3     U(I)=...
S4   END DO
     communication request of:
          T(IC) for values of CAP1_LOW-2 ≤ IC ≤ CAP1_LOW-1
          U(IC) for values of IC=CAP1_LOW-1
S5   B=...
```

Figure B.58: Example illustrating that only two communications are needed to satisfy the communication requests of the small example shown in Figure B.49. The communication requests were first migrated up the control flow graph using the predominator tree (to execute before statement S5), where it was then possible to merge them.

For multi-dimensional arrays, as well as testing the partitioned component of the requested data, a vector space comparison test is used to ensure that other usage index components are subsets as well [25]. Therefore, as a result of merging the control sets for the requested data (by obtaining the union of several requests), the startup latency cost is reduced, due to the reduced number of communications needed to communicate the data.

B.9.1.6 Generation Of Communications

Communications are generated for variables based on the communication control sets that have been migrated and then merged at a precise location in the code. The communication direction is dependent on the requested data, where a communication in the lower direction (CAP_LEFT, CAP_UP, CAP_BACK, etc) will involve the lower processor partition range limit, and a communication in the upper direction (CAP_RIGHT, CAP_DOWN, CAP_FORTH, etc) will involve the upper processor partition range limit. The type of communication used depends on the number of conditions in the simplified control set. For example, an Exchange is used if there are 2 conditions (as was the case for the example in Figure B.49), a Send/Receive is used if there are at least 3 conditions, and a Broadcast is used if the control set cannot be normalised. If statement S16 in Figure B.49 were the only statement requesting data from a neighbouring processor then the simplified control statement would have 3 conditions, since there would not be any other requests to merge with. A check is made on the number of contiguous items of data in memory, where the communication is buffered if necessary (with loops generated around the communication in certain cases). Requests for different variables may have been migrated and merged at the same location, where a communication will be generated for each variable separately, as illustrated in Figure B.59.

```
CALL CAP_EXCHANGE(T(CAP1_LOW-2), T(CAP1_HIGH-1),2,2,CAP_LEFT)
CALL CAP_EXCHANGE(U(CAP1-LOW-1),U(CAP1_HIGH),1,2,CAP_LEFT)
```

Figure B.59: The communications that are required to satisfy the requests made in Figure B.49 (which will be executed after the assignment of the communicated data, before statement S5).

The user can generate communications for their code in the Code Generator window (shown in Figure B.46), where they can choose from a number of different communication types to generate, and they can select the minimum width of the halo region (MIN_SLABS). Once the communications have been generated the user is able to browse through these using the Communications Browser (Figure B.60). The user can examine the different types of communications that have been generated for each routine, investigating why a

communication has been generated by looking at the assignment and usage statements of the communicated data in the 'Why Communication?' window (Figure B.61), as well as examining the key dependence (see Figure B.53). Once the user is satisfied with the generated communications, they can proceed to complete the parallelisation of their code.

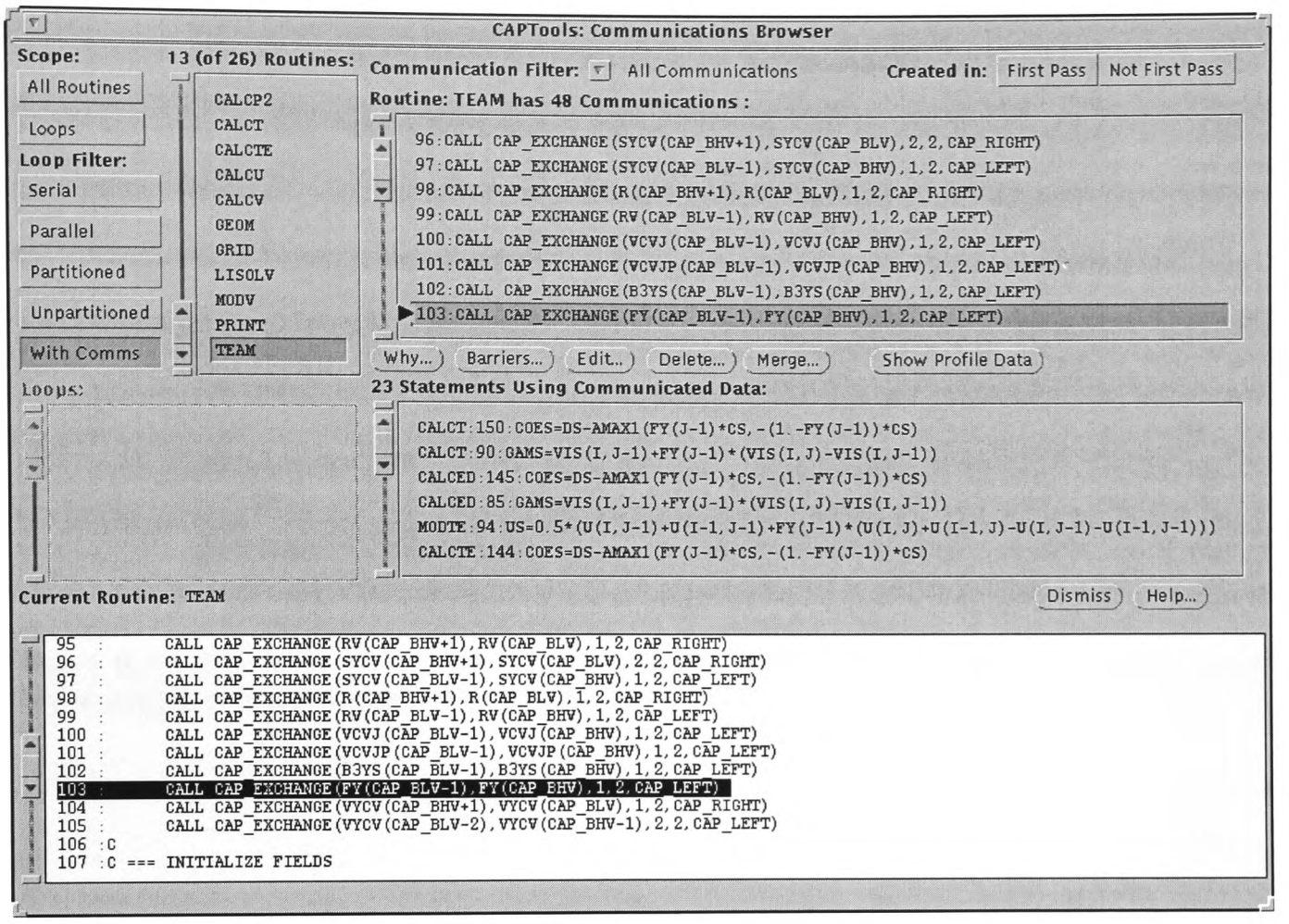


Figure B.60: The Communications Browser window, used to examine generated communications of the current partition within CAPTools.

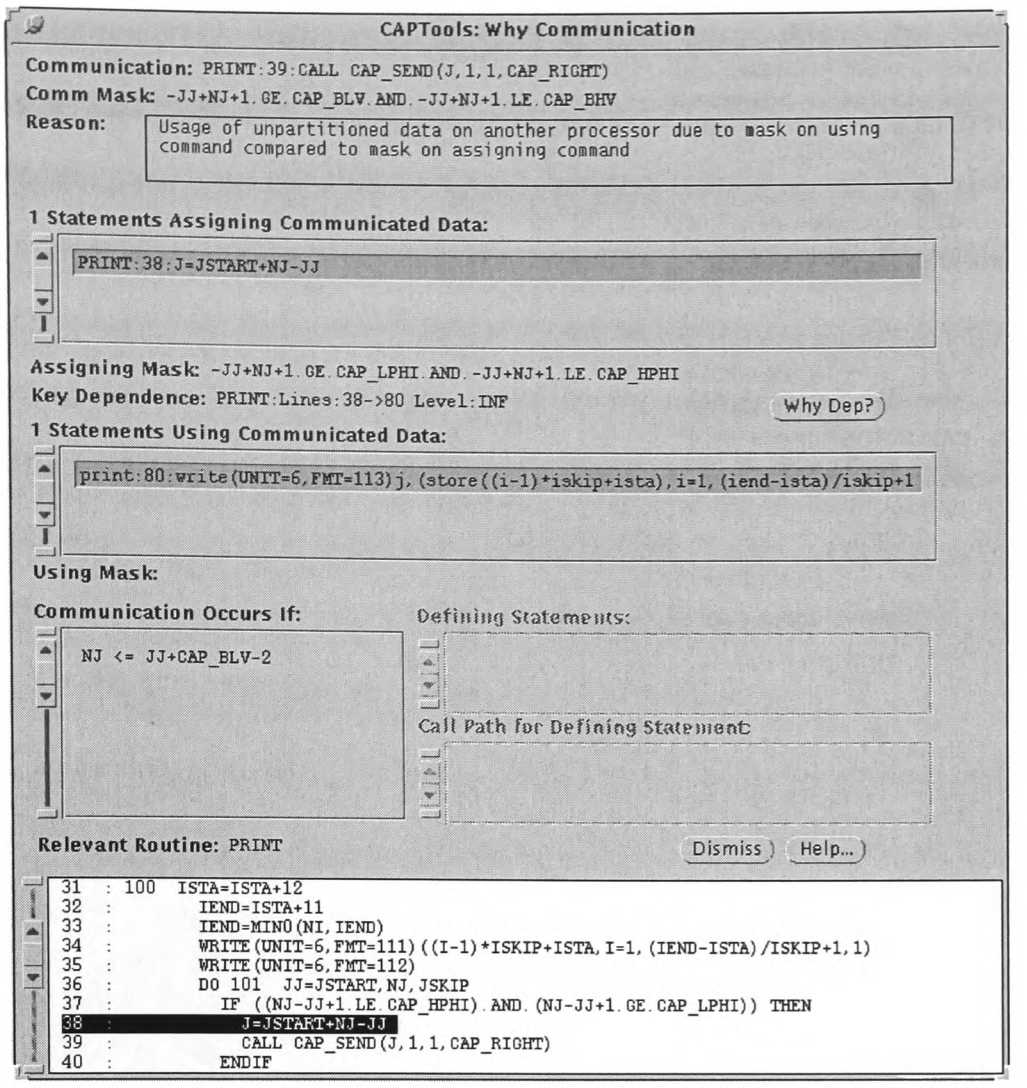


Figure B.61: The Why Communication window which can be used to examine the reasons why a selected communication was generated.

B.9.2 Communication Data Structures

All of the requests are stored in the RECEIVE data structure of the communication statement, shown in Figure B.62, where the usage SYMBOL refers to the requested data. For example, when migrating the communication request control sets for the example in Figure B.49, the RECEIVE record for that location (whose generated requests would be executed before statement S5) would initially list several entries for T, and one entry for U. Details such as the individual control set (CONTROL), the partitioned component (EXPRESSION, and PARTITION), the mask (MASKEXPRESSION, and MASKPARTITION), the communication direction (DIRECTION), and details on the actual commands requesting the communication (COMMANDLIST) with associated assigners for unpartitioned data listed in ASSIGNLIST, are stored for each entry. Additionally,

DEFROUTE is used to store the migration call path from the request to the communication location.

When the communication request entries relating to the same data are merged after migration, their CONTROL sets and COMMANDLISTs are merged, where details relating to the subset request are absorbed into the other request. In Figure B.49 for example, after merging the entries containing (IC=CAP1_LOW-1 and IC=NI-1) and (IC=CAP1_LOW-1) , requested in statements S16 and S8 respectively, the merged entry's CONTROL set would then contain (IC=CAP1_LOW-1), where its COMMANDLIST would contain S8 and S16. Other entries would be merged in a similar manner.

```
RECEIVE=RECORD
  SYMBOL:PTABLE;
  CONTROL:ANDLIST;
  EXPRESSION,MASKEXPRESSION:PLINKER;
  PARTITION,MASKPARTITION:PPARTITION;
  STATEMENT:PSTATEMENT;
  COMMAND:PCOMMAND;
  COMMANDLIST,ASSIGNLIST:PCOMMSCOMMANDLIST;
  NESTING:PLOOPS;
  DEFROUTE:PLISTROUTINE;
  DIRECTION,SEND,RECEIVE:BOOLEAN;
  NEXT:PRECEIVE;
  END;
```

Figure B.62: The RECEIVE data structure that is stored for every command.

The COMMSCOMMANDLIST record stores information about commands involved in requesting communications (both usage command, and for unpartitioned data the assignment command causing a dependence). The fields in the communication requester command record are shown in Figure B.63.

```
COMMSCOMMANDLIST=RECORD
  ROUTINE:PROUTINE;
  COMMAND:PCOMMAND;
  DEFROUTE:PLISTROUTINE;
  NEXT:PCOMMSCOMMANDLIST;
  END;
```

Figure B.63: The COMMSCOMMANDLIST data structure.

The basic parse tree structure for a CAP_SEND communication call can be seen in Figure B.64, for which similar tree structures are used for the other CAPTools communications utilities.

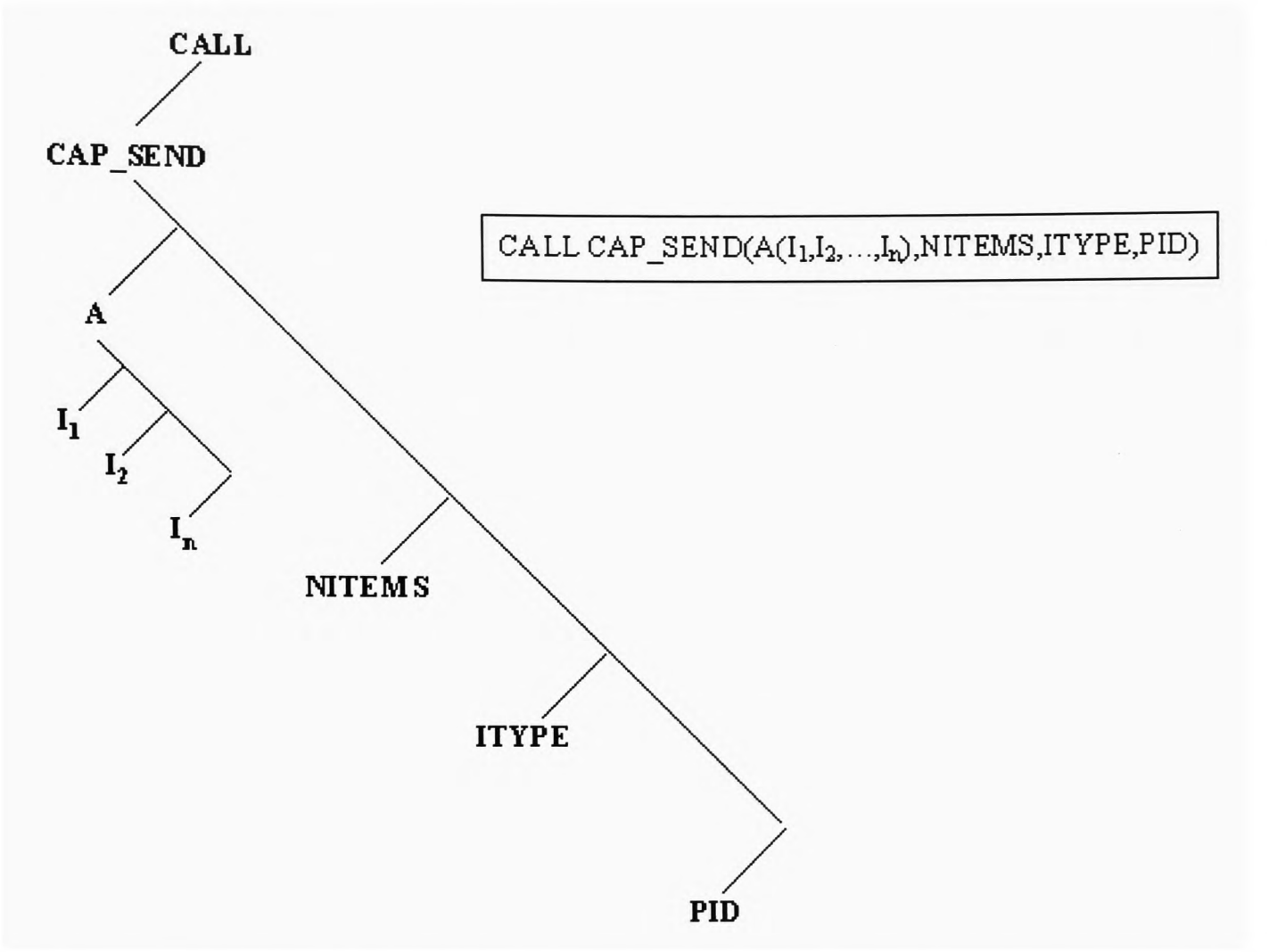


Figure B.64: Tree structure for the CAP_SEND communication call utility in CAPTools.

B.10 Reduced Memory

CAPTools provides the user with the option to reduce the total memory usage per processor [113]. Storing the whole data array can be unnecessary, especially if a processor is not going to be operating on the whole range. If for example A(10000) was partitioned, where a single processor was to operate on A(8101:8200), reduced memory enables the array to be reduced in size such that each processor only need to store the data that may be worked upon. In Figure B.65 for example, instead of each processor having to store the whole array, each processor would now only store the region of data that they operate upon (assign within), their own halo regions, and other entries.

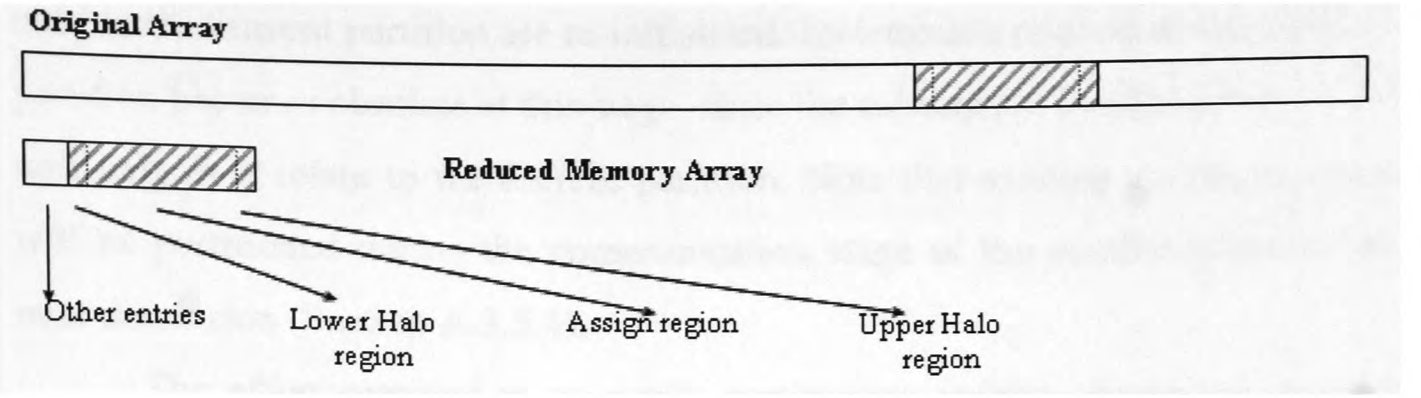


Figure B.65: Processors store entire array unless Reduced Memory option is selected.

Figure B.66 illustrates the need for each processor to store their assign region, their halo region, and the extreme boundaries of the data array B. Values of K=1 and K=NK are required on all processors, where this data can be stored under other entries.

```
...
   A(I,J,K)=B(I,J,K)+B(I,J,K-1)+B(I,J,K+1)+B(I,J,1)+B(I,J,NK)
...
```

Figure B.66: Example illustrating the need to store the assign region, halo regions, and the extreme boundaries (in other entries), when applying Reduced Memory.

B.11 Partition Next Dimension (Multi-Dimensional Partitioning)

CAPTools is a semi-automatic tool that enables the user to partition more than one dimension, where the user is able to select the option (in the Code Generator window, Figure B.46) to partition another dimension after completing the masking and communication generation phases. The same parallelisation process is undertaken for the partitioning of all further dimensions. An example of a 3D array that has been partitioned first in the I direction, then J direction, and finally the K direction, is shown in Figure A.3, where the processor partition range limits are given for the 1D, 2D, and 3D, partitioning.

Once the user has decided to partition another dimension then CAPTools sets itself up ready to store information relating to the new partition, and so information relating to the previous partition is destroyed as the data structures

used in the current partition are re-initialised. Information relating to the previous partition becomes obsolete at this stage, since the subsequent parallelisation stages will only ever relate to the current partition. Note that existing communications will be partitioned during the communication stage of the parallelisation of the next dimension (Section A.3.3.4).

The effort required in manually partitioning another dimension may be disenchanting to the user, such that the parallelisation of a code is dependent upon the attitude of the user. There is little additional effort required to partition another dimension when using CAPTools, enabling codes to have multi-dimensional partitioning rather than the usual 1D partition. The user is no longer restricted by the number of partitions, as CAPTools can be used effortlessly to generate multi-dimensional parallel code, which would have taken a lot longer to implement manually.

B.12 Generating And Saving The Final Parallel Code

This is the last stage in the parallelisation process, where a CAPTools parallel version of the input serial code is generated. Having performed a dependence analysis, the user was able to partition their code, generate execution control masks, and calculate and generate the necessary communication calls (repeating this process for further dimensions). The generated parallel code should still be recognisable to the user, enabling them to maintain and optimise their code without difficulty.

It is at this stage, having selected the option to 'Generate Final Parallel Code' (Figure B.46), that the parallel code is physically created. Up until this stage it was a virtual parallel code, where the internal data structures representing the parallel code could be viewed using the CAPTools browser windows, indicating how the virtual parallel code may be altered before creating the parallel code.

B.13 Summary

CAPTools is a semi-automatic parallelisation tool that allows the user to interactively generate a multi-dimensional parallel version of their serial code. Accurate dependence analysis is a vital component that enables the user to partition data in a number of dimensions, changing the loop limits from global into local limits, and inserting communication statements. The generated code is portable due to the use of the generic CAPLib message passing calls that permit the user to compile and run the same code on a number of platforms [112].

Dependence analysis is an important stage in the automatic parallelisation of any code, examining the dependencies that exist between different statements within the code. A good analysis can enable a very good data partition, and can lead to efficient communications being generated.

This Appendix provided a detailed insight into the algorithms and data structures that are used by CAPTools. An understanding of how CAPTools operates will be vital in understanding the automatic implementation of DLB within a CAPTools generated parallel code.

Appendix C Automatically Generated DLB Parallel Version Of The FAB Code

This Appendix contains the automatically generated DLB parallel version of the FAB code (Section 5.11), excluding the actual DLB utility routines. Note that a manual implementation of DLB within the parallel version of this code was not undertaken.

```
      PROGRAM PARALLELFAB
      INTEGER CAP_UP2,CAP_DOWN2
      PARAMETER (CAP_UP2=-3,CAP_DOWN2=-4)
      INTEGER CAP_LEFT,CAP_RIGHT
      PARAMETER (CAP_LEFT=-1,CAP_RIGHT=-2)
      REAL TIME,TL,DT,CON1,RIN,Z(500),R(500),TOLD(500,500),RHO,CP,TNEW(
     +500,500),HCF(0:3),WKSP(500,500),SK(500,500),HFLX(500,500),WTH,HGT,
     +T0,ZF,ZL,RF,RL,FACX,FACY,TMBY(0:3),Q0,KO,CON2,KON(500,500),TIMES
      CHARACTER ANS
      INTEGER GMOPT,IN,JN,PRIN,TCOUNT,PMON,RSTRT,INUM,IMON,JMON
      COMMON /GEOM/IN,JN,GMOPT,RIN,WTH,HGT,IMON,JMON,FACX,FACY
      COMMON /PROP/RHO,CP,KO,Q0,KON
      COMMON /SOLUTN/DT,CON1,PMON
      INTEGER CAP_BLTNEW,CAP_BHTNEW
      COMMON /CAP_RANGE/CAP_BLTNEW,CAP_BHTNEW
      INTEGER CAP_ICOUNT
      INTEGER I
      INTEGER CAP_PROCNUM,CAP_NPROC
      COMMON /CAP_TOOLS/CAP_PROCNUM,CAP_NPROC
      INTEGER CAP2_BLTNEW,CAP2_BHTNEW
      COMMON /CAP2_RANGE/CAP2_BLTNEW,CAP2_BHTNEW
      COMMON /CAP_GLOBALVARS/CAP_J65SOLVER
      INTEGER CAP_J65SOLVER
      INTEGER CAP_J
      INTEGER CAP_DLB_STAG_DIM
      COMMON /CAP_DLB_STAG_DIM/CAP_DLB_STAG_DIM
      CALL CAP_INIT()
      CALL CAP_DLB_SETALLNEIGHBOURS
      CAP_DLB_STAG_DIM=2
      IF (CAP_PROCNUM.EQ.1)OPEN(UNIT=10,FILE='INPUT.DAT')
      IF (CAP_PROCNUM.EQ.1)REWIND(10)
      RIN=0.0
      IF (CAP_PROCNUM.EQ.1)PRINT *,
     +'DO YOU WISH TO USE EXISTING PROBLEM SETUP (Y OR N)?'
      IF (CAP_PROCNUM.EQ.1)READ(UNIT=*,FMT=111)ANS
      CALL CAP_RECEIVE(ANS,1*LEN(ANS),6,CAP_UP2)
      CALL CAP_SEND(ANS,1*LEN(ANS),6,CAP_DOWN2)
      CALL CAP_RECEIVE(ANS,1*LEN(ANS),6,CAP_LEFT)
      CALL CAP_SEND(ANS,1*LEN(ANS),6,CAP_RIGHT)
  111 FORMAT(A)
      RSTRT=1
      CALL CONDUC(RSTRT,TIME,TL,DT,PMON,Z,R,TOLD,TNEW,T0,CON1,PRIN,HCF,
     +TMBY,WKSP,SK,HFLX,CAP_BLTNEW,CAP_BHTNEW,CAP2_BLTNEW,CAP2_BHTNEW)
      CALL CAP_EXCHANGE(Z(CAP2_BHTNEW+1),Z(CAP2_BLTNEW),1,2,CAP_DOWN2)
      CALL CAP_EXCHANGE(Z(CAP2_BLTNEW-1),Z(CAP2_BHTNEW),1,2,CAP_UP2)
        DO CAP_J=max(2,CAP_BLTNEW),MIN(JN-1,CAP_BHTNEW)
        CALL CAP_EXCHANGE(WKSP(CAP2_BHTNEW+1,CAP_J),WKSP(CAP2_BLTNEW,
     +  CAP_J),1,2,CAP_DOWN2)
        ENDDO
        DO CAP_J=max(2,CAP_BLTNEW),MIN(JN-1,CAP_BHTNEW)
        IF (((IN.LE.CAP2_BHTNEW).AND.(IN.GE.CAP2_BLTNEW)).AND.(IN-1.LT.
     +  CAP2_BLTNEW)) THEN
          CALL CAP_SEND(TNEW(IN,CAP_J),1,2,CAP_UP2)
        ENDIF
        IF ((IN.GT.CAP2_BHTNEW).AND.(IN-1.LE.CAP2_BHTNEW)) THEN
          CALL CAP_RECEIVE(TNEW(IN,CAP_J),1,2,CAP_DOWN2)
        ENDIF
        IF ((IN.GT.CAP2_BHTNEW).AND.(IN-1.LT.CAP2_BLTNEW)) THEN
          CALL CAP_SEND(TNEW(IN,CAP_J),1,2,CAP_UP2)
        ENDIF
        ENDDO
        DO CAP_J=max(2,CAP_BLTNEW),MIN(JN-1,CAP_BHTNEW)
        IF (((1.LE.CAP2_BHTNEW).AND.(1.GE.CAP2_BLTNEW)).AND.(2.GT.
```

```
     +  CAP2_BHTNEW)) THEN
          CALL CAP_SEND(TNEW(1,CAP_J),1,2,CAP_DOWN2)
        ENDIF
        IF ((1.LT.CAP2_BLTNEW).AND.(2.GE.CAP2_BLTNEW)) THEN
          CALL CAP_RECEIVE(TNEW(1,CAP_J),1,2,CAP_UP2)
        ENDIF
        IF ((1.LT.CAP2_BLTNEW).AND.(2.GT.CAP2_BHTNEW)) THEN
          CALL CAP_SEND(TNEW(1,CAP_J),1,2,CAP_DOWN2)
        ENDIF
        ENDDO
       CALL CAP_EXCHANGE(R(CAP_BHTNEW+1),R(CAP_BLTNEW),1,2,CAP_RIGHT)
       CALL CAP_EXCHANGE(R(CAP_BLTNEW-1),R(CAP_BHTNEW),1,2,CAP_LEFT)
       CALL CAP_DLB_EXCHANGE(SK(CAP2_BLTNEW,CAP_BHTNEW+1),SK(CAP2_BLTNEW,
      +CAP_BLTNEW),CAP2_BHTNEW-CAP2_BLTNEW+1,CAP2_BLTNEW,1,CAP2_BLTNEW,
      +CAP2_BHTNEW,2,CAP_RIGHT)
C
C      CREATE INPUT DATA FILE
C
       IF (CAP_PROCNUM.EQ.1)REWIND(10)
       CON2=CON1
       IF (CAP_PROCNUM.EQ.1)WRITE(UNIT=10,FMT=*)TIME,TL,DT,PMON
       IF (CAP_PROCNUM.EQ.1)WRITE(UNIT=10,FMT=*)GMOPT,RIN
       IF (CAP_PROCNUM.EQ.1)WRITE(UNIT=10,FMT=*)IN,JN,IMON,JMON,HGT,WTH
       IF (CAP_PROCNUM.EQ.1)WRITE(UNIT=10,FMT=*)FACX,FACY
       IF (CAP_PROCNUM.EQ.1)WRITE(UNIT=10,FMT=*)T0
       IF (CAP_PROCNUM.EQ.1)WRITE(UNIT=10,FMT=*)CON2,PRIN
       IF (CAP_PROCNUM.EQ.1)WRITE(UNIT=10,FMT=*)HCF(0),HCF(1),HCF(2),HCF(
      +3)
       IF (CAP_PROCNUM.EQ.1)WRITE(UNIT=10,FMT=*)TMBY(0),TMBY(1),TMBY(2),
      +TMBY(3)
       IF (CAP_PROCNUM.EQ.1)WRITE(UNIT=10,FMT=*)KO
       IF (CAP_PROCNUM.EQ.1)WRITE(UNIT=10,FMT=*)RHO,CP,Q0
C
C      PRINT INITIAL FIELDS.
C
       IF ((2.LE.CAP_BHTNEW).AND.(2.GE.CAP_BLTNEW)) THEN
         RF=(R(1)+R(2))*0.5
         CALL CAP_SEND(RF,1,2,CAP_RIGHT)
       ENDIF
       IF (2.LT.CAP_BLTNEW) THEN
         CALL CAP_RECEIVE(RF,1,2,CAP_LEFT)
       ENDIF
       IF (2.LT.CAP_BLTNEW) THEN
         CALL CAP_SEND(RF,1,2,CAP_RIGHT)
       ENDIF
       IF ((JN-1.LE.CAP_BHTNEW).AND.(JN-1.GE.CAP_BLTNEW)) THEN
         RL=(R(JN)+R(JN-1))*0.5
         CALL CAP_SEND(RL,1,2,CAP_LEFT)
       ENDIF
       IF (JN-1.GT.CAP_BHTNEW) THEN
         CALL CAP_RECEIVE(RL,1,2,CAP_RIGHT)
       ENDIF
       IF (JN-1.GT.CAP_BHTNEW) THEN
         CALL CAP_SEND(RL,1,2,CAP_LEFT)
       ENDIF
       IF ((2.LE.CAP2_BHTNEW).AND.(2.GE.CAP2_BLTNEW)) THEN
         ZF=(Z(1)+Z(2))*0.5
         CALL CAP_SEND(ZF,1,2,CAP_DOWN2)
       ENDIF
       IF (2.LT.CAP2_BLTNEW) THEN
         CALL CAP_RECEIVE(ZF,1,2,CAP_UP2)
       ENDIF
       IF (2.LT.CAP2_BLTNEW) THEN
         CALL CAP_SEND(ZF,1,2,CAP_DOWN2)
       ENDIF
       IF ((IN-1.LE.CAP2_BHTNEW).AND.(IN-1.GE.CAP2_BLTNEW)) THEN
         ZL=(Z(IN)+Z(IN-1))*0.5
         CALL CAP_SEND(ZL,1,2,CAP_UP2)
       ENDIF
       IF (IN-1.GT.CAP2_BHTNEW) THEN
         CALL CAP_RECEIVE(ZL,1,2,CAP_DOWN2)
       ENDIF
       IF (IN-1.GT.CAP2_BHTNEW) THEN
         CALL CAP_SEND(ZL,1,2,CAP_UP2)
       ENDIF
       IF (CAP_PROCNUM.EQ.1)PRINT *,'Z GRID'
         DO I=1,IN-1+1-2
         IF (I+1.GT.CAP2_BHTNEW)CALL CAP_DLB_RECEIVE(Z(I+2-1),1,I+1,0,
      +  CAP2_BLTNEW,CAP2_BHTNEW,2,CAP_DOWN2)
         IF (I+1.GE.CAP2_BLTNEW)CALL CAP_DLB_SEND(Z(I+2-1),1,I+1,0,
      +  CAP2_BLTNEW,CAP2_BHTNEW,2,CAP_UP2)
         ENDDO
       IF (CAP_PROCNUM.EQ.1)PRINT *,ZF,(Z(I+2-1),I=1,IN-1+1-2),ZL
       IF (CAP_PROCNUM.EQ.1)PRINT *,'R GRID'
         DO I=1,JN-1+1-2
         IF (I+1.GT.CAP_BHTNEW) THEN
           CALL CAP_RECEIVE(R(I+2-1),1,2,CAP_RIGHT)
```

```
        ENDIF
        IF (I+1.GE.CAP_BLTNEW) THEN
          CALL CAP_SEND(R(I+2-1),1,2,CAP_LEFT)
        ENDIF
        ENDDO
      IF (CAP_PROCNUM.EQ.1)PRINT *,RF,(R(I+2-1),I=1,JN-1+1-2),RL
      IF (CAP_PROCNUM.EQ.1)PRINT *,'INITIAL TEMPERATURE.'
C
C     START TIMER
C
      T1=SECOND()
C
C        MAIN LOOP CONTROLS NUMBER OF TIME STEPS.
C
   20   IF (TIME.LT.TL) THEN
          TIME=TIME+DT
          CALL SOLVER(TIME,Z,R,TOLD,TNEW,HFLX,WKSP,SK,CAP_BLTNEW,
     +    CAP_BHTNEW,CAP2_BLTNEW,CAP2_BHTNEW)
            DO 30  J=max(1,CAP_BLTNEW),MIN(JN,CAP_BHTNEW)
              DO 30  I=max(1,CAP2_BLTNEW),MIN(IN,CAP2_BHTNEW),1
              TOLD(I,J)=TNEW(I,J)
   30         CONTINUE
            CONTINUE
          GOTO 20
      ELSE
C
C       STOP TIMER
C
        T2=SECOND()-T1
        IF (CAP_PROCNUM.EQ.1)PRINT *,'ITERATION:',ISWEEP,'  TIME:',T2
        IF (CAP_PROCNUM.EQ.1)PRINT *
      ENDIF
  222 FORMAT(6(1X,E12.4))
      IF (CAP_PROCNUM.EQ.1)CLOSE(10)
      CALL CAP_FINISH()
      END


      SUBROUTINE CONDUC(RSTRT,TIME,TL,DT,PMON,Z,R,TOLD,TNEW,T0,CON1,PRIN
     +,HCF,TMBY,WKSP,SK,HFLX,CAP_LHFLX,CAP_HHFLX,CAP2_LHFLX,CAP2_HHFLX)
      INTEGER CAP_UP2,CAP_DOWN2
      PARAMETER (CAP_UP2=-3,CAP_DOWN2=-4)
      INTEGER CAP_LEFT,CAP_RIGHT
      PARAMETER (CAP_LEFT=-1,CAP_RIGHT=-2)
C
C     THIS ROUTINE SETS UP THE CONDUCTION PARAMETERS
C
      COMMON /GEOM/IN,JN,GMOPT,RIN,WTH,HGT,IMON,JMON,FACX,FACY
      COMMON /PROP/RHO,CP,KO,Q0,KON
      REAL RIN,WTH,GHT,FACX,FACY,RHO,CP,KO,Q0,TIME,TL,DT,Z(500),R(500),
     +TNEW(500,500),TOLD(500,500),CON1,HCF(0:3),TMBY(0:3),WKSP(500,500),
     +SK(500,500),T0,HFLX(500,500),KON(500,500)
      INTEGER IN,JN,GMOPT,IMON,JMON,RSTRT,PMON,PRIN
C
C     SET UP NECESSARY DATA
C
      INTEGER CAP_LHFLX,CAP_HHFLX
      COMMON /CAP_RANGE/CAP_BLTNEW,CAP_BHTNEW
      INTEGER CAP_BLTNEW,CAP_BHTNEW
      INTEGER CAP2_LHFLX,CAP2_HHFLX
      COMMON /CAP2_RANGE/CAP2_BLTNEW,CAP2_BHTNEW
      INTEGER CAP2_BLTNEW,CAP2_BHTNEW
      COMMON /CAP_GLOBALVARS/CAP_J65SOLVER
      INTEGER CAP_J65SOLVER
      CALL SETTEMP(TIME,TL,DT,PMON,RSTRT)
      CALL GEOMET(Z,R,RSTRT,CAP_LHFLX,CAP_HHFLX,CAP2_LHFLX,CAP2_HHFLX)
      CALL CAP_EXCHANGE(Z(CAP2_LHFLX-1),Z(CAP2_HHFLX),1,2,CAP_UP2)
      CALL CAP_EXCHANGE(R(CAP_LHFLX-1),R(CAP_HHFLX),1,2,CAP_LEFT)
      CALL TEMPER(TOLD,TNEW,T0,RSTRT,CAP_LHFLX,CAP_HHFLX,CAP2_LHFLX,
     +CAP2_HHFLX)
      CALL CONVER(CON1,PRIN,RSTRT)
      CALL HTCOEF(Z,R,HCF,TMBY,TNEW,RSTRT,CAP_LHFLX,CAP_HHFLX,CAP2_LHFLX
     +,CAP2_HHFLX)
      CALL PROPS(HCF,WKSP,SK,HFLX,Z,R,RSTRT,CAP_LHFLX,CAP_HHFLX,
     +CAP2_LHFLX,CAP2_HHFLX)
      RETURN
      END


      SUBROUTINE SETTEMP(TIME,TL,DT,PMON,RSTRT)
      INTEGER CAP_UP2,CAP_DOWN2
      PARAMETER (CAP_UP2=-3,CAP_DOWN2=-4)
      INTEGER CAP_LEFT,CAP_RIGHT
      PARAMETER (CAP_LEFT=-1,CAP_RIGHT=-2)
C
C     THIS ROUTINE DEFINES A PROBLEM SPECIFICATION.
C
```

```
      REAL TIME,TL,DT
      INTEGER PMON,ITIME,RSTRT
      COMMON /CAP_RANGE/CAP_BLTNEW,CAP_BHTNEW
      INTEGER CAP_BLTNEW,CAP_BHTNEW
      INTEGER CAP_PROCNUM,CAP_NPROC
      COMMON /CAP_TOOLS/CAP_PROCNUM,CAP_NPROC
      COMMON /CAP2_RANGE/CAP2_BLTNEW,CAP2_BHTNEW
      INTEGER CAP2_BLTNEW,CAP2_BHTNEW
      COMMON /CAP_GLOBALVARS/CAP_J65SOLVER
      INTEGER CAP_J65SOLVER
      IF (RSTRT.EQ.0) THEN
      ELSE
        IF (CAP_PROCNUM.EQ.1)READ(UNIT=10,FMT=*)TIME,TL,DT,PMON
        CALL CAP_RECEIVE(TIME,1,2,CAP_UP2)
        CALL CAP_SEND(TIME,1,2,CAP_DOWN2)
        CALL CAP_RECEIVE(TL,1,2,CAP_UP2)
        CALL CAP_SEND(TL,1,2,CAP_DOWN2)
        CALL CAP_RECEIVE(DT,1,2,CAP_UP2)
        CALL CAP_SEND(DT,1,2,CAP_DOWN2)
        CALL CAP_RECEIVE(PMON,1,1,CAP_UP2)
        CALL CAP_SEND(PMON,1,1,CAP_DOWN2)
        CALL CAP_RECEIVE(TIME,1,2,CAP_LEFT)
        CALL CAP_SEND(TIME,1,2,CAP_RIGHT)
        CALL CAP_RECEIVE(TL,1,2,CAP_LEFT)
        CALL CAP_SEND(TL,1,2,CAP_RIGHT)
        CALL CAP_RECEIVE(DT,1,2,CAP_LEFT)
        CALL CAP_SEND(DT,1,2,CAP_RIGHT)
        CALL CAP_RECEIVE(PMON,1,1,CAP_LEFT)
        CALL CAP_SEND(PMON,1,1,CAP_RIGHT)
      ENDIF
      RETURN
      END


      SUBROUTINE CONVER(CON1,PRIN,RSTRT)
      INTEGER CAP_UP2,CAP_DOWN2
      PARAMETER (CAP_UP2=-3,CAP_DOWN2=-4)
      INTEGER CAP_LEFT,CAP_RIGHT
      PARAMETER (CAP_LEFT=-1,CAP_RIGHT=-2)
C
C     THIS ROUTINE CHECKS FOR CONVERGENCE
C
      REAL RIN,CON1,WTH,HGT,FACX,FACY
      INTEGER IN,JN,GMOPT,IMON,JMON,PRIN,RSTRT
      COMMON /GEOM/IN,JN,GMOPT,RIN,WTH,HGT,IMON,JMON,FACX,FACY
C
C     SET CONVERGENCE AND PRINT PARAMETERS.
C
      COMMON /CAP_RANGE/CAP_BLTNEW,CAP_BHTNEW
      INTEGER CAP_BLTNEW,CAP_BHTNEW
      INTEGER CAP_PROCNUM,CAP_NPROC
      COMMON /CAP_TOOLS/CAP_PROCNUM,CAP_NPROC
      COMMON /CAP2_RANGE/CAP2_BLTNEW,CAP2_BHTNEW
      INTEGER CAP2_BLTNEW,CAP2_BHTNEW
      COMMON /CAP_GLOBALVARS/CAP_J65SOLVER
      INTEGER CAP_J65SOLVER
      IF (RSTRT.EQ.0) THEN
      ELSE
        IF (CAP_PROCNUM.EQ.1)READ(UNIT=10,FMT=*)CON1,PRIN
        CALL CAP_RECEIVE(CON1,1,2,CAP_UP2)
        CALL CAP_SEND(CON1,1,2,CAP_DOWN2)
        CALL CAP_RECEIVE(PRIN,1,1,CAP_UP2)
        CALL CAP_SEND(PRIN,1,1,CAP_DOWN2)
        CALL CAP_RECEIVE(CON1,1,2,CAP_LEFT)
        CALL CAP_SEND(CON1,1,2,CAP_RIGHT)
        CALL CAP_RECEIVE(PRIN,1,1,CAP_LEFT)
        CALL CAP_SEND(PRIN,1,1,CAP_RIGHT)
      ENDIF
      RETURN
      END


      SUBROUTINE GEOMET(Z,R,RSTRT,CAP_LR,CAP_HR,CAP2_LZ,CAP2_HZ)
      INTEGER CAP_UP2,CAP_DOWN2
      PARAMETER (CAP_UP2=-3,CAP_DOWN2=-4)
      INTEGER CAP_LEFT,CAP_RIGHT
      PARAMETER (CAP_LEFT=-1,CAP_RIGHT=-2)
C
C     THIS ROUTINE SETS UP THE GRID GEOMETRY SPECIFICATION AS (IN,JN)
C
      REAL DZ,VAL,WTH,RIN,HGT,Z(500),R(500),FACX,FACY
      INTEGER GMOPT,IN,JN,LOC,IMON,JMON,RSTRT
      CHARACTER*18 LAB,XLAB0,XLAB1,YLAB0,YLAB1,XNAM0,XNAM1,YNAM0,YNAM1
      CHARACTER ANS,XLABMON,YLABMON
      PARAMETER (FIXVAL=1.0E+10,ADIABAT=1.0E-10)
      PARAMETER (XLAB0='X',XLAB1='Z',YLAB0='Y',YLAB1='R',XNAM0=
     +'HEIGHT    ',XNAM1='RADIAL THICKNESS',YNAM0='WIDTH',YNAM1=
```

```
     +'LENGTH OF CYLINDER')
      COMMON /GEOM/IN,JN,GMOPT,RIN,WTH,HGT,IMON,JMON,FACX,FACY
      INTEGER CAP_LR,CAP_HR
      COMMON /CAP_RANGE/CAP_BLTNEW,CAP_BHTNEW
      INTEGER CAP_BLTNEW,CAP_BHTNEW
      INTEGER CAP_PROCNUM,CAP_NPROC
      COMMON /CAP_TOOLS/CAP_PROCNUM,CAP_NPROC
      INTEGER CAP2_LZ,CAP2_HZ
      COMMON /CAP2_RANGE/CAP2_BLTNEW,CAP2_BHTNEW
      INTEGER CAP2_BLTNEW,CAP2_BHTNEW
      COMMON /CAP_GLOBALVARS/CAP_J65SOLVER
      INTEGER CAP_J65SOLVER
      IF (CAP_PROCNUM.EQ.1)READ(UNIT=10,FMT=*)GMOPT,RIN
      CALL CAP_RECEIVE(GMOPT,1,1,CAP_UP2)
      CALL CAP_SEND(GMOPT,1,1,CAP_DOWN2)
      CALL CAP_RECEIVE(RIN,1,2,CAP_UP2)
      CALL CAP_SEND(RIN,1,2,CAP_DOWN2)
      CALL CAP_RECEIVE(GMOPT,1,1,CAP_LEFT)
      CALL CAP_SEND(GMOPT,1,1,CAP_RIGHT)
      CALL CAP_RECEIVE(RIN,1,2,CAP_LEFT)
      CALL CAP_SEND(RIN,1,2,CAP_RIGHT)
      IF (CAP_PROCNUM.EQ.1)READ(UNIT=10,FMT=*)IN,JN,IMON,JMON,HGT,WTH
      CALL CAP_RECEIVE(IN,1,1,CAP_UP2)
      CALL CAP_SEND(IN,1,1,CAP_DOWN2)
      CALL CAP_RECEIVE(JN,1,1,CAP_UP2)
      CALL CAP_SEND(JN,1,1,CAP_DOWN2)
      CALL CAP_RECEIVE(IMON,1,1,CAP_UP2)
      CALL CAP_SEND(IMON,1,1,CAP_DOWN2)
      CALL CAP_RECEIVE(JMON,1,1,CAP_UP2)
      CALL CAP_SEND(JMON,1,1,CAP_DOWN2)
      CALL CAP_RECEIVE(HGT,1,2,CAP_UP2)
      CALL CAP_SEND(HGT,1,2,CAP_DOWN2)
      CALL CAP_RECEIVE(WTH,1,2,CAP_UP2)
      CALL CAP_SEND(WTH,1,2,CAP_DOWN2)
      CALL CAP_RECEIVE(IN,1,1,CAP_LEFT)
      CALL CAP_SEND(IN,1,1,CAP_RIGHT)
      CALL CAP_RECEIVE(JN,1,1,CAP_LEFT)
      CALL CAP_SEND(JN,1,1,CAP_RIGHT)
      CALL CAP_RECEIVE(IMON,1,1,CAP_LEFT)
      CALL CAP_SEND(IMON,1,1,CAP_RIGHT)
      CALL CAP_RECEIVE(JMON,1,1,CAP_LEFT)
      CALL CAP_SEND(JMON,1,1,CAP_RIGHT)
      CALL CAP_RECEIVE(HGT,1,2,CAP_LEFT)
      CALL CAP_SEND(HGT,1,2,CAP_RIGHT)
      CALL CAP_RECEIVE(WTH,1,2,CAP_LEFT)
      CALL CAP_SEND(WTH,1,2,CAP_RIGHT)
      CALL CAP_SETUPDPART(1,IN,CAP2_BLTNEW,CAP2_BHTNEW,2)
      CALL CAP_DLB_SETUPLIMITS(CAP2_BLTNEW,CAP2_BHTNEW,2)
      CALL CAP_SETUPDPART(1,JN,CAP_BLTNEW,CAP_BHTNEW,1)
      CALL CAP_DLB_SETUPLIMITS(CAP_BLTNEW,CAP_BHTNEW,1)
      IF (CAP_PROCNUM.EQ.1)READ(UNIT=10,FMT=*)FACX,FACY
      CALL CAP_RECEIVE(FACX,1,2,CAP_UP2)
      CALL CAP_SEND(FACX,1,2,CAP_DOWN2)
      CALL CAP_RECEIVE(FACY,1,2,CAP_UP2)
      CALL CAP_SEND(FACY,1,2,CAP_DOWN2)
      CALL CAP_RECEIVE(FACX,1,2,CAP_LEFT)
      CALL CAP_SEND(FACX,1,2,CAP_RIGHT)
      CALL CAP_RECEIVE(FACY,1,2,CAP_LEFT)
      CALL CAP_SEND(FACY,1,2,CAP_RIGHT)
      IF (ABS(FACX-1.0).LE.ADIABAT) THEN
        DZ=WTH/(IN-2)
      ELSE
        F1=FACX**(IN-2)
        DZ=2.0*WTH*(1.0-FACX)/(1.0+FACX-F1-F1*FACX)
      ENDIF
C
C        SET UP THE Y GRID.
C
      IF (ABS(FACY-1.0).LE.ADIABAT) THEN
C
C          UNIFORM GRID.
C
        DR=HGT/(JN-2)
      ELSE
        F1=FACY**(JN-2)
        DR=2.0*HGT*(1.0-FACY)/(1.0+FACY-F1-F1*FACY)
      ENDIF
C
C        EXTERNAL NODES.
C
      IF ((1.LE.CAP2_HZ).AND.(1.GE.CAP2_LZ)) THEN
        Z(1)=-DZ/2.0
      ENDIF
      IF ((1.LE.CAP_HR).AND.(1.GE.CAP_LR)) THEN
        R(1)=-DR/2.0
        IF (GMOPT.GT.0) THEN
          R(1)=R(1)+RIN
```

```
        ENDIF
      ENDIF
C
C        CALCULATE REMAINDER OF NODES.
C
      DO 10  I=2,IN,1
        IF (((I-1.LE.CAP2_HZ).AND.(I-1.GE.CAP2_LZ)).AND.(I.GT.CAP2_HZ))
     +   THEN
          CALL CAP_SEND(Z(I-1),1,2,CAP_DOWN2)
        ENDIF
        IF ((I-1.LT.CAP2_LZ).AND.(I.GE.CAP2_LZ)) THEN
          CALL CAP_RECEIVE(Z(I-1),1,2,CAP_UP2)
        ENDIF
        IF ((I-1.LT.CAP2_LZ).AND.(I.GT.CAP2_HZ)) THEN
          CALL CAP_SEND(Z(I-1),1,2,CAP_DOWN2)
        ENDIF
        IF ((I.LE.CAP2_HZ).AND.(I.GE.CAP2_LZ)) THEN
          Z(I)=Z(I-1)+DZ
        ENDIF
        DZ=DZ*FACX
   10 CONTINUE
      DO 20  J=2,JN,1
        IF (((J-1.LE.CAP_HR).AND.(J-1.GE.CAP_LR)).AND.(J.GT.CAP_HR))
     +   THEN
          CALL CAP_SEND(R(J-1),1,2,CAP_RIGHT)
        ENDIF
        IF ((J-1.LT.CAP_LR).AND.(J.GE.CAP_LR)) THEN
          CALL CAP_RECEIVE(R(J-1),1,2,CAP_LEFT)
        ENDIF
        IF ((J-1.LT.CAP_LR).AND.(J.GT.CAP_HR)) THEN
          CALL CAP_SEND(R(J-1),1,2,CAP_RIGHT)
        ENDIF
        IF ((J.LE.CAP_HR).AND.(J.GE.CAP_LR)) THEN
          R(J)=R(J-1)+DR
        ENDIF
        DR=DR*FACY
   20 CONTINUE
      RETURN
      END


      SUBROUTINE TEMPER(TOLD,TNEW,T0,RSTRT,CAP_LTNEW,CAP_HTNEW,
     +CAP2_LTNEW,CAP2_HTNEW)
      INTEGER CAP_UP2,CAP_DOWN2
      PARAMETER (CAP_UP2=-3,CAP_DOWN2=-4)
      INTEGER CAP_LEFT,CAP_RIGHT
      PARAMETER (CAP_LEFT=-1,CAP_RIGHT=-2)
C
C        THIS ROUTINE WILL SET UP TNEW AND TOLD AS AN ARRAY.
C
      REAL TOLD(500,500),TNEW(500,500),VAL,T0,FACX,FACY,WTH,HGT,RIN
      INTEGER IL,IU,JL,JU,IN,JN,GS,RSTRT,IMON,JMON,GMOPT
      CHARACTER ANS
      COMMON /GEOM/IN,JN,GMOPT,RIN,WTH,HGT,IMON,JMON,FACX,FACY
      INTEGER CAP_LTNEW,CAP_HTNEW
      COMMON /CAP_RANGE/CAP_BLTNEW,CAP_BHTNEW
      INTEGER CAP_BLTNEW,CAP_BHTNEW
      INTEGER CAP_PROCNUM,CAP_NPROC
      COMMON /CAP_TOOLS/CAP_PROCNUM,CAP_NPROC
      INTEGER CAP2_LTNEW,CAP2_HTNEW
      COMMON /CAP2_RANGE/CAP2_BLTNEW,CAP2_BHTNEW
      INTEGER CAP2_BLTNEW,CAP2_BHTNEW
      COMMON /CAP_GLOBALVARS/CAP_J65SOLVER
      INTEGER CAP_J65SOLVER
      IF (RSTRT.EQ.0) THEN
      ELSE
        IF (CAP_PROCNUM.EQ.1)READ(UNIT=10,FMT=*)T0
        CALL CAP_RECEIVE(T0,1,2,CAP_UP2)
        CALL CAP_SEND(T0,1,2,CAP_DOWN2)
        CALL CAP_RECEIVE(T0,1,2,CAP_LEFT)
        CALL CAP_SEND(T0,1,2,CAP_RIGHT)
      ENDIF
      DO 30  J=MAX(2,CAP_LTNEW),MIN(JN-1,CAP_HTNEW),1
        DO 30  I=MAX(2,CAP2_LTNEW),MIN(IN-1,CAP2_HTNEW),1
          TOLD(I,J)=T0
          TNEW(I,J)=T0
   30   CONTINUE
      CONTINUE
C
C        NON-UNIFORMATY LOOP
C
  100 FORMAT(A)
      RETURN
      END


      SUBROUTINE HTCOEF(Z,R,HCF,TMBY,TNEW,RSTRT,CAP_LR,CAP_HR,CAP2_LTNEW
```

```
     +,CAP2_HTNEW)
      INTEGER CAP_UP2,CAP_DOWN2
      PARAMETER (CAP_UP2=-3,CAP_DOWN2=-4)
      INTEGER CAP_LEFT,CAP_RIGHT
      PARAMETER (CAP_LEFT=-1,CAP_RIGHT=-2)
C
C        THIS ROUTINE WILL SET UP HEAT TRANSFER COEFFICIENTS AND
C     AND BOUNDARY TEMPERATURES.
C
      CHARACTER*18 LAB00,LAB01,LAB02,LAB03,LAB10,LAB11,LAB
      CHARACTER ANS
      REAL HCF(0:3),TMBY(0:3),RIN,TNEW(500,500),Z(500),R(500),WTH,HGT,
     +FACX,FACY
      INTEGER GMOPT,IN,JN,LOC,IMON,JMON,RSTRT
      PARAMETER (FIXVAL=1.0E+10,ADIABAT=1.0E-10,LAB00='NORTH',LAB01=
     +'SOUTH',LAB02='WEST',LAB03='EAST',LAB10='EXTERNAL',LAB11=
     +'INTERNAL')
      COMMON /GEOM/IN,JN,GMOPT,RIN,WTH,HGT,IMON,JMON,FACX,FACY
      INTEGER CAP_LR,CAP_HR
      COMMON /CAP_RANGE/CAP_BLTNEW,CAP_BHTNEW
      INTEGER CAP_BLTNEW,CAP_BHTNEW
      INTEGER CAP_PROCNUM,CAP_NPROC
      COMMON /CAP_TOOLS/CAP_PROCNUM,CAP_NPROC
      INTEGER CAP2_LTNEW,CAP2_HTNEW
      COMMON /CAP2_RANGE/CAP2_BLTNEW,CAP2_BHTNEW
      INTEGER CAP2_BLTNEW,CAP2_BHTNEW
      COMMON /CAP_GLOBALVARS/CAP_J65SOLVER
      INTEGER CAP_J65SOLVER
      IF (RSTRT.EQ.0) THEN
  100   FORMAT(A)
        IF ((ANS.NE.'N').AND.(ANS.NE.'n')) THEN
        ELSE
C
C            FIX THE BOUNDARY TEMPERATURE AS FOLLOWS :-
C            IF HCF .LT. ADIABAT THEN APPLY SYMMETRY I.E. TMBY = TP
C          IF ADIABAT .GT. HCF .LT. FIXVAL APPLY A FIXED HEAT TRANSFER
C            IF HCF .GT. FIXVAL FIX TMBY.
C
        ENDIF
      ELSE
        IF (CAP_PROCNUM.EQ.1)READ(UNIT=10,FMT=*)HCF(0),HCF(1),HCF(2),HCF
     +  (3)
        CALL CAP_RECEIVE(HCF(0),1,2,CAP_UP2)
        CALL CAP_SEND(HCF(0),1,2,CAP_DOWN2)
        CALL CAP_RECEIVE(HCF(1),1,2,CAP_UP2)
        CALL CAP_SEND(HCF(1),1,2,CAP_DOWN2)
        CALL CAP_RECEIVE(HCF(2),1,2,CAP_UP2)
        CALL CAP_SEND(HCF(2),1,2,CAP_DOWN2)
        CALL CAP_RECEIVE(HCF(3),1,2,CAP_UP2)
        CALL CAP_SEND(HCF(3),1,2,CAP_DOWN2)
        CALL CAP_RECEIVE(HCF(0),1,2,CAP_LEFT)
        CALL CAP_SEND(HCF(0),1,2,CAP_RIGHT)
        CALL CAP_RECEIVE(HCF(1),1,2,CAP_LEFT)
        CALL CAP_SEND(HCF(1),1,2,CAP_RIGHT)
        CALL CAP_RECEIVE(HCF(2),1,2,CAP_LEFT)
        CALL CAP_SEND(HCF(2),1,2,CAP_RIGHT)
        CALL CAP_RECEIVE(HCF(3),1,2,CAP_LEFT)
        CALL CAP_SEND(HCF(3),1,2,CAP_RIGHT)
        IF (CAP_PROCNUM.EQ.1)READ(UNIT=10,FMT=*)TMBY(0),TMBY(1),TMBY(2),
     +  TMBY(3)
        CALL CAP_RECEIVE(TMBY(0),1,2,CAP_UP2)
        CALL CAP_SEND(TMBY(0),1,2,CAP_DOWN2)
        CALL CAP_RECEIVE(TMBY(1),1,2,CAP_UP2)
        CALL CAP_SEND(TMBY(1),1,2,CAP_DOWN2)
        CALL CAP_RECEIVE(TMBY(2),1,2,CAP_UP2)
        CALL CAP_SEND(TMBY(2),1,2,CAP_DOWN2)
        CALL CAP_RECEIVE(TMBY(3),1,2,CAP_UP2)
        CALL CAP_SEND(TMBY(3),1,2,CAP_DOWN2)
        CALL CAP_RECEIVE(TMBY(0),1,2,CAP_LEFT)
        CALL CAP_SEND(TMBY(0),1,2,CAP_RIGHT)
        CALL CAP_RECEIVE(TMBY(1),1,2,CAP_LEFT)
        CALL CAP_SEND(TMBY(1),1,2,CAP_RIGHT)
        CALL CAP_RECEIVE(TMBY(2),1,2,CAP_LEFT)
        CALL CAP_SEND(TMBY(2),1,2,CAP_RIGHT)
        CALL CAP_RECEIVE(TMBY(3),1,2,CAP_LEFT)
        CALL CAP_SEND(TMBY(3),1,2,CAP_RIGHT)
        IF (((JN.LE.CAP_HR).AND.(JN.GE.CAP_LR)).OR.((1.LE.CAP_HR).AND.(1
     +  .GE.CAP_LR))) THEN
            DO 40  I=MAX(1,CAP2_LTNEW),MIN(IN,CAP2_HTNEW),1
            IF ((JN.LE.CAP_HR).AND.(JN.GE.CAP_LR)) THEN
              TNEW(I,JN)=TMBY(0)
            ENDIF
            IF ((1.LE.CAP_HR).AND.(1.GE.CAP_LR)) THEN
              TNEW(I,1)=TMBY(1)
            ENDIF
   40       CONTINUE
        ENDIF
```

```
          IF (((1.LE.CAP2_HTNEW).AND.(1.GE.CAP2_LTNEW)).OR.((IN.LE.
     +    CAP2_HTNEW).AND.(IN.GE.CAP2_LTNEW))) THEN
              DO 50  J=MAX(1,CAP_LR),MIN(JN,CAP_HR)
              IF ((1.LE.CAP2_HTNEW).AND.(1.GE.CAP2_LTNEW)) THEN
                TNEW(1,J)=TMBY(2)
              ENDIF
              IF ((IN.LE.CAP2_HTNEW).AND.(IN.GE.CAP2_LTNEW)) THEN
                TNEW(IN,J)=TMBY(3)
              ENDIF
   50         CONTINUE
          ENDIF
        ENDIF
        RETURN
        END


        SUBROUTINE PROPS(HCF,WKSP,SK,HFLX,Z,R,RSTRT,CAP_LHFLX,CAP_HHFLX,
       +CAP2_LHFLX,CAP2_HHFLX)
        INTEGER CAP_UP2,CAP_DOWN2
        PARAMETER (CAP_UP2=-3,CAP_DOWN2=-4)
        INTEGER CAP_LEFT,CAP_RIGHT
        PARAMETER (CAP_LEFT=-1,CAP_RIGHT=-2)
C
C          THIS ROUTINE ALLOWS THE INPUT OF PHYSICAL PROPERTIES AND ALSO
C       CALCULATES THE  CONDUCTIVITIES TO WEST AND SOUTH OF NODE.
C
        COMMON /GEOM/IN,JN,GMOPT,RIN,WTH,HGT,IMON,JMON,FACX,FACY
        COMMON /PROP/RHO,CP,KO,Q0,K
        REAL RHO,CP,WKSP(500,500),SK(500,500),K(500,500),KO,R(500),Z(500),
       +HCF(0:3),HFLX(500,500),WTH,HGT,FACX,FACY,Q0,RIN
        INTEGER GMOPT,IMON,JMON,RSTRT
        CHARACTER ANS
        INTEGER CAP_LHFLX,CAP_HHFLX
        COMMON /CAP_RANGE/CAP_BLTNEW,CAP_BHTNEW
        INTEGER CAP_BLTNEW,CAP_BHTNEW
        INTEGER CAP_PROCNUM,CAP_NPROC
        COMMON /CAP_TOOLS/CAP_PROCNUM,CAP_NPROC
        INTEGER CAP2_LHFLX,CAP2_HHFLX
        COMMON /CAP2_RANGE/CAP2_BLTNEW,CAP2_BHTNEW
        INTEGER CAP2_BLTNEW,CAP2_BHTNEW
        COMMON /CAP_GLOBALVARS/CAP_J65SOLVER
        INTEGER CAP_J65SOLVER
        INTEGER CAP_J
        IF ((JN.LE.CAP_HHFLX).AND.(JN.GE.CAP_LHFLX)) THEN
          DRN=(R(JN)-R(JN-1))*0.5*HCF(0)
        ENDIF
        IF ((1.LE.CAP_HHFLX).AND.(1.GE.CAP_LHFLX)) THEN
          DR1=(R(2)-R(1))*0.5*HCF(1)
        ENDIF
        IF ((IN.LE.CAP2_HHFLX).AND.(IN.GE.CAP2_LHFLX)) THEN
          DZN=(Z(IN)-Z(IN-1))*0.5*HCF(3)
        ENDIF
        IF ((1.LE.CAP2_HHFLX).AND.(1.GE.CAP2_LHFLX)) THEN
          DZ1=(Z(2)-Z(1))*0.5*HCF(2)
        ENDIF
        IF (((1.LE.CAP_HHFLX).AND.(1.GE.CAP_LHFLX)).OR.((JN.LE.CAP_HHFLX)
       +.AND.(JN.GE.CAP_LHFLX))) THEN
            DO 10  I=max(1,CAP2_LHFLX),MIN(IN,CAP2_HHFLX),1
            IF ((1.LE.CAP_HHFLX).AND.(1.GE.CAP_LHFLX)) THEN
              K(I,1)=DR1
            ENDIF
            IF ((JN.LE.CAP_HHFLX).AND.(JN.GE.CAP_LHFLX)) THEN
              K(I,JN)=DRN
            ENDIF
   10       CONTINUE
        ENDIF
        IF (((1.LE.CAP2_HHFLX).AND.(1.GE.CAP2_LHFLX)).OR.((IN.LE.
       +CAP2_HHFLX).AND.(IN.GE.CAP2_LHFLX))) THEN
            DO 20  J=max(1,CAP_LHFLX),MIN(JN,CAP_HHFLX)
            IF ((1.LE.CAP2_HHFLX).AND.(1.GE.CAP2_LHFLX)) THEN
              K(1,J)=DZ1
            ENDIF
            IF ((IN.LE.CAP2_HHFLX).AND.(IN.GE.CAP2_LHFLX)) THEN
              K(IN,J)=DZN
            ENDIF
   20       CONTINUE
        ENDIF
        IF (RSTRT.EQ.0) THEN
        ELSE
          IF (CAP_PROCNUM.EQ.1)READ(UNIT=10,FMT=*)KO
          CALL CAP_RECEIVE(KO,1,2,CAP_UP2)
          CALL CAP_SEND(KO,1,2,CAP_DOWN2)
          CALL CAP_RECEIVE(KO,1,2,CAP_LEFT)
          CALL CAP_SEND(KO,1,2,CAP_RIGHT)
        ENDIF
          DO 30  J=max(2,CAP_LHFLX),MIN(JN-1,CAP_HHFLX),1
            DO 30  I=max(2,CAP2_LHFLX),MIN(IN-1,CAP2_HHFLX),1
```

```
            K(I,J)=KO
   30       CONTINUE
          CONTINUE
          DO CAP_J=max(2,CAP_LHFLX),MIN(JN,CAP_HHFLX)
          CALL CAP_EXCHANGE(K(CAP2_LHFLX-1,CAP_J),K(CAP2_HHFLX,CAP_J),1,2,
        +  CAP_UP2)
          ENDDO
         CALL CAP_DLB_EXCHANGE(K(CAP2_LHFLX,CAP_LHFLX-1),K(CAP2_LHFLX,
        +CAP_HHFLX),CAP2_HHFLX-CAP2_LHFLX+1,CAP2_LHFLX,1,CAP2_LHFLX,
        +CAP2_HHFLX,2,CAP_LEFT)
  300 FORMAT(6(1X,E12.4))
C
C          SET UP NON-UNIFORMITIES
C
  100   FORMAT(A)
          DO 60  J=max(2,CAP_LHFLX),MIN(JN,CAP_HHFLX),1
            DO 60  I=max(2,CAP2_LHFLX),MIN(IN,CAP2_HHFLX),1
            WKSP(I,J)=2.0*K(I-1,J)*K(I,J)/(K(I-1,J)+K(I,J))/(Z(I)-Z(I-1))
            SK(I,J)=2.0*K(I,J-1)*K(I,J)/(K(I,J-1)+K(I,J))/(R(J)-R(J-1))
            IF (GMOPT.EQ.1) THEN
              SK(I,J)=SK(I,J)*(R(J)+R(J-1))*0.5
            ENDIF
   60       CONTINUE
          CONTINUE
        IF (RSTRT.EQ.0) THEN
        ELSE
          IF (CAP_PROCNUM.EQ.1)READ(UNIT=10,FMT=*)RHO,CP,Q0
          CALL CAP_RECEIVE(RHO,1,2,CAP_UP2)
          CALL CAP_SEND(RHO,1,2,CAP_DOWN2)
          CALL CAP_RECEIVE(CP,1,2,CAP_UP2)
          CALL CAP_SEND(CP,1,2,CAP_DOWN2)
          CALL CAP_RECEIVE(Q0,1,2,CAP_UP2)
          CALL CAP_SEND(Q0,1,2,CAP_DOWN2)
          CALL CAP_RECEIVE(RHO,1,2,CAP_LEFT)
          CALL CAP_SEND(RHO,1,2,CAP_RIGHT)
          CALL CAP_RECEIVE(CP,1,2,CAP_LEFT)
          CALL CAP_SEND(CP,1,2,CAP_RIGHT)
          CALL CAP_RECEIVE(Q0,1,2,CAP_LEFT)
          CALL CAP_SEND(Q0,1,2,CAP_RIGHT)
        ENDIF
          DO 70  J=max(1,CAP_LHFLX),MIN(JN,CAP_HHFLX)
            DO 70  I=max(1,CAP2_LHFLX),MIN(IN,CAP2_HHFLX),1
            HFLX(I,J)=Q0
   70       CONTINUE
          CONTINUE
        RETURN
        END


        SUBROUTINE SOLVER(TIME,Z,R,TOLD,TNEW,HFLX,WKSP,SK,CAP_LHFLX,
       +CAP_HHFLX,CAP2_LA,CAP2_HA)
        INTEGER CAP_UP2,CAP_DOWN2
        PARAMETER (CAP_UP2=-3,CAP_DOWN2=-4)
        INTEGER CAP_LEFT,CAP_RIGHT
        PARAMETER (CAP_LEFT=-1,CAP_RIGHT=-2)
C
C       THIS ROUTINE SETS UP HEAT EQUATION. THIS IS DONE IN
C       THE FOLLOWING WAY:-
C         A(I)*TNEW(I-1,J) + D(I)*TNEW(I,J) + C(I)*TNEW(I+1,J) = B(I)
C
C         WHERE B(I) CONTAINS TNEW(I,J+1),TNEW(I,J-1),TOLD(I,J),
C         HFLX(I,J) TERMS.
C
C         IT ALSO CONTROLS THE SWEEP ACTION OF THE SOLUTION PROCEDURE,
C         (WHICH SWEEPS J LINES).
C         THE TDMA ALGORITHM IS USED TO SOLVE EACH LINE, A CALL TO THE
C         SUBROUTINE TDMA IS MADE. FINALLY A CALL TO THE ROUTINE
C         RESIDUAL IS MADE WHICH DETERMINES THE RESIDUAL
C         AT THE END OF EACH SWEEP.
C
C
        REAL DT,RHO,CP,RIN,CON1,TIME,FAC,RESID,Z(500),R(500),RBAR,KO,DR,DZ
       +,TNEW(500,500),TOLD(500,500),HFLX(500,500),A(500),B(500),C(500),D(
       +500),DIFF,DIFF0,WKSP(500,500),SK(500,500),WTH,HGT,FACX,FACY,Q0,KON
       +(500,500),LSWEEP(500),RESIDJ
        INTEGER IN,JN,GMOPT,PRIN,isweep,MSWEEP,IMON,JMON
        COMMON /GEOM/IN,JN,GMOPT,RIN,WTH,HGT,IMON,JMON,FACX,FACY
        COMMON /TDMARR/A,B,C,D
        COMMON /PROP/RHO,CP,KO,Q0,KON
        COMMON /SOLUTN/DT,CON1,PRIN
C
C          SWEEP COUNTER.
C
        INTEGER CAP_LHFLX,CAP_HHFLX
        COMMON /CAP_RANGE/CAP_BLTNEW,CAP_BHTNEW
        INTEGER CAP_BLTNEW,CAP_BHTNEW
        EXTERNAL CAP_RADD
```

```
      REAL CAP_RADD
      INTEGER CAP_PROCNUM,CAP_NPROC
      COMMON /CAP_TOOLS/CAP_PROCNUM,CAP_NPROC
      INTEGER CAP2_LA,CAP2_HA
      COMMON /CAP2_RANGE/CAP2_BLTNEW,CAP2_BHTNEW
      INTEGER CAP2_BLTNEW,CAP2_BHTNEW
      COMMON /CAP_GLOBALVARS/CAP_J65SOLVER
      INTEGER CAP_J65SOLVER
      INTEGER CAP_DLB_ITER,CAP_DLB_REBAL_ITER
      REAL CAP_DLB_WALL_TIME,CAP_DLB_COMM_TIME,CAP_DLB_COMP_TIME,
     +CAP_DLB_MAX_TIME,CAP_DLB_REBAL_TIME,CAP_DLB_PREV_REBAL_TIME
      LOGICAL CAP_DLB_PERFORM_REBAL,CAP_DLB_MIGRATE_DIM(6)
      CAP_DLB_PREV_REBAL_TIME=0.0
      CAP_DLB_REBAL_TIME=0.0
      CAP_DLB_MAX_TIME=0.0
      CAP_DLB_COMP_TIME=0.0
      CAP_DLB_COMM_TIME=0.0
      CAP_DLB_WALL_TIME=0.0
      isweep=0
C
C    ON FIRST ENTRY TO THE SUBROUTINE SET MAX SWEEP AND FAC
C
      if (TIME*0.99.LT.DT) THEN
C
C          MAX SWEEP NUMBER AT A GIVEN TIME,IF DT IS NOT CONSTANT SET UP
C       OUTSIDE IF STATEMENT
C
        MSWEEP=5000
        FAC=RHO*CP/DT
      ENDIF
C
C        THIS IS THE MAIN LOOP, IT CONTROLS NUMBER OF SWEEPS.
C
   40 CALL CAP_DLB_DECIDE(CAP_DLB_WALL_TIME,CAP_DLB_COMM_TIME,
     +CAP_DLB_COMP_TIME,CAP_DLB_MAX_TIME,CAP_DLB_PERFORM_REBAL,
     +CAP_DLB_ITER,CAP_DLB_REBAL_ITER,CAP_DLB_REBAL_TIME)
      if (CAP_DLB_PERFORM_REBAL) THEN
        CALL CAP_DLB_START_REBAL(CAP_DLB_REBAL_TIME,CAP_DLB_COMP_TIME,
     +  CAP_DLB_MAX_TIME,CAP_DLB_PREV_REBAL_TIME,CAP_DLB_MIGRATE_DIM)
        if (CAP_DLB_MIGRATE_DIM(1)) THEN
          CALL CAP_DLB_MIGRATE(HFLX(1,1),1,500,1,1,1,1,1,1,1,1,1,1,1
     +    ,1,1,1,1,2,1)
        if (CAP_DLB_MIGRATE_DIM(1)) THEN
          CALL CAP_DLB_MIGRATE(HFLX(1,1),1,500,1,1,1,1,1,1,1,1,1,1,1
     +    ,1,1,1,1,2,1)
          CALL CAP_DLB_MIGRATE(KON(1,1),1,500,1,1,1,1,1,1,1,1,1,1,1,
     +    1,1,1,1,2,1)
          CALL CAP_MIGRATE(R(1),1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,1)
          CALL CAP_DLB_MIGRATE(SK(1,1),1,500,1,1,1,1,1,1,1,1,1,1,1
     +    ,1,1,1,1,2,1)
          CALL CAP_DLB_MIGRATE(TNEW(1,1),1,500,1,1,1,1,1,1,1,1,1,1,1
     +    ,1,1,1,1,2,1)
          CALL CAP_DLB_MIGRATE(TOLD(1,1),1,500,1,1,1,1,1,1,1,1,1,1,1
     +    ,1,1,1,1,2,1)
          CALL CAP_DLB_MIGRATE(WKSP(1,1),1,500,1,1,1,1,1,1,1,1,1,1,1
     +    ,1,1,1,1,2,1)
          CALL CAP_DLB_REASSIGNLOWHIGH(CAP_LHFLX,CAP_HHFLX,1)
        ENDIF
        if (CAP_DLB_MIGRATE_DIM(2)) THEN
          CALL CAP_MIGRATE(A(1),1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,2)
          CALL CAP_MIGRATE(B(1),1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,2)
          CALL CAP_MIGRATE(C(1),1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,2)
          CALL CAP_MIGRATE(D(1),1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,2)
          CALL CAP_MIGRATE(HFLX(1,CAP_LHFLX),1,1,500,CAP_HHFLX-
     +    CAP_LHFLX+1,1,1,1,1,1,1,1,1,1,1,1,2,2)
          CALL CAP_MIGRATE(KON(1,CAP_LHFLX),1,1,500,CAP_HHFLX-
     +    CAP_LHFLX+1,1,1,1,1,1,1,1,1,1,1,1,2,2)
          CALL CAP_MIGRATE(LSWEEP(1),1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2
     +    ,2)
          CALL CAP_MIGRATE(SK(1,CAP_LHFLX),1,1,500,CAP_HHFLX-
     +    CAP_LHFLX+1,1,1,1,1,1,1,1,1,1,1,1,2,2)
          CALL CAP_MIGRATE(TNEW(1,CAP_LHFLX),1,1,500,CAP_HHFLX-
     +    CAP_LHFLX+1,1,1,1,1,1,1,1,1,1,1,1,2,2)
          CALL CAP_MIGRATE(TOLD(1,CAP_LHFLX),1,1,500,CAP_HHFLX-
     +    CAP_LHFLX+1,1,1,1,1,1,1,1,1,1,1,1,2,2)
          CALL CAP_MIGRATE(WKSP(1,CAP_LHFLX),1,1,500,CAP_HHFLX-
     +    CAP_LHFLX+1,1,1,1,1,1,1,1,1,1,1,1,2,2)
          CALL CAP_MIGRATE(Z(1),1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,2)
          CALL CAP_DLB_REASSIGNLOWHIGH(CAP2_LA,CAP2_HA,2)
        ENDIF
        if (CAP_DLB_MIGRATE_DIM(1).OR.CAP_DLB_MIGRATE_DIM(2)) THEN
          CALL CAP_DLB_NEW2OLD_LIMITS
          CALL CAP_EXCHANGE(Z(CAP2_BHTNEW+1),Z(CAP2_BLTNEW),1,2,
     +    CAP_DOWN2)
          CALL CAP_EXCHANGE(Z(CAP2_BLTNEW-1),Z(CAP2_BHTNEW),1,2,
     +    CAP_UP2)
          DO CAP_J=2,JN-1
            CALL CAP_EXCHANGE(WKSP(CAP2_BHTNEW+1,CAP_J),WKSP(
     +      CAP2_BLTNEW,CAP_J),1,2,CAP_DOWN2)
          ENDDO
```

```
            CALL CAP_EXCHANGE(R(CAP_BHTNEW+1),R(CAP_BLTNEW),1,2,CAP_RIGHT)
            CALL CAP_EXCHANGE(R(CAP_BLTNEW-1),R(CAP_BHTNEW),1,2,CAP_LEFT)
            CALL CAP_DLB_EXCHANGE(SK(CAP2_BLTNEW,CAP_BHTNEW+1),SK(
     +      CAP2_BLTNEW,CAP_BLTNEW),CAP2_BHTNEW-CAP2_BLTNEW+1,
     +      CAP2_BLTNEW,1,CAP2_BLTNEW,CAP2_BHTNEW,2,CAP_RIGHT)
            CALL CAP_DLB_STOP_REBAL(CAP_DLB_REBAL_TIME,CAP_DLB_ITER,
     +      CAP_DLB_REBAL_ITER)
          ELSE
            CAP_DLB_REBAL_TIME=CAP_DLB_PREV_REBAL_TIME
          ENDIF
        ENDIF
        CALL CAP_DLB_START_TIMER(CAP_DLB_WALL_TIME,CAP_DLB_COMM_TIME)
        CALL CAP_DLB_EXCHANGE(TNEW(CAP2_LA,CAP_BLTNEW-1),TNEW(CAP2_LA,
     +CAP_BHTNEW),CAP2_HA-CAP2_LA+1,CAP2_LA,1,CAP2_LA,CAP2_HA,2,
     +CAP_LEFT)
        CALL CAP_DLB_EXCHANGE(TNEW(CAP2_LA,CAP_BHTNEW+1),TNEW(CAP2_LA,
     +CAP_BLTNEW),CAP2_HA-CAP2_LA+1,CAP2_LA,1,CAP2_LA,CAP2_HA,2,
     +CAP_RIGHT)
C
C         IF MAX SWEEP REACHED PRINT OUT RESULTS AND QUIT
C
       if (isweep.LE.MSWEEP) THEN
C
C           START TO SWEEP LINES VISITING EACH J LINE IN DOMAIN ONCE.
C
          TOP=0.0
          BOT=0.0
          DO 30  J=max(2,CAP_BLTNEW),MIN(JN-1,CAP_BHTNEW),1
            CAP_J65SOLVER=J
            if (GMOPT.eq.0) THEN
              RBAR=1.0
            ELSE
              RBAR=R(J)
            ENDIF
            DR=(R(J+1)-R(J-1))/2.0*RBAR
C
C             CONSTRUCT COEFF.(GAUSS-SEIDAL ITERATION IMPLEMENTED
C           SO MUST USE LATEST VALUES OF TNEW(I,J-1) FOR EACH LINE
C           CALCULATION.)
C
            DO 10  I=max(2,CAP2_LA),MIN(IN-1,CAP2_HA),1
              LSWEEP(I)=TNEW(I,J)
              DZ=(Z(I+1)-Z(I-1))*0.5
              A(I)=WKSP(I,J)/DZ
              C(I)=WKSP(I+1,J)/DZ
              D(I)=-(A(I)+C(I)+FAC+(SK(I,J+1)+SK(I,J))/DR)
              B(I)=TOLD(I,J)*FAC+HFLX(I,J)
              B(I)=-(B(I)+(TNEW(I,J+1)*SK(I,J+1)+TNEW(I,J-1)*SK(I,J))/DR)
   10       CONTINUE
            CALL CAP_EXCHANGE(C(CAP2_BLTNEW-1),C(CAP2_BHTNEW),1,2,CAP_UP2)
            CALL TDMA(TNEW,IN,IN-1,J,CAP_LHFLX,CAP_HHFLX,CAP2_LA,CAP2_HA)
            CALL RESIDUAL(LSWEEP,TNEW,IN-1,RESIDJ,J,JN-1,TOP,BOT,
     +      CAP_LHFLX,CAP_HHFLX,CAP2_LA,CAP2_HA)
   30     CONTINUE
          CALL CAP_DCOMMUTATIVE(BOT,2,CAP_RADD,CAP_LEFT)
          CALL CAP_DCOMMUTATIVE(TOP,2,CAP_RADD,CAP_LEFT)
          TOP=SQRT(TOP)
          BOT=SQRT(BOT)+1.0
          RESIDJ=TOP/BOT
          RESID=RESIDJ
          isweep=isweep+1
C
C           IS RESIDUAL PRINT OUT REQUIRED.
C
          if (mod(isweep,10).eq.0) THEN
            IF (CAP_PROCNUM.EQ.1)PRINT *,'RESIDUAL = ',RESID,isweep
          ENDIF
          if (RESID.GT.CON1) THEN
            GOTO 40
          ELSE
            IF (CAP_PROCNUM.EQ.1)PRINT *,'ITERATIONS:',isweep
            RETURN
          ENDIF
        ENDIF
        RETURN
        END


        SUBROUTINE TDMA(Y,IN,IN1,J,CAP_LY,CAP_HY,CAP2_LA,CAP2_HA)
        INTEGER CAP_UP2,CAP_DOWN2
        PARAMETER (CAP_UP2=-3,CAP_DOWN2=-4)
        INTEGER CAP_LEFT,CAP_RIGHT
        PARAMETER (CAP_LEFT=-1,CAP_RIGHT=-2)
C
C       SOLVES A TRIDIAGONAL SYSTEM OF EQUATIONS USING THOMAS ALGORITHM
C
        REAL Y(500,500),A(500),B(500),C(500),D(500),M1
```

```
      INTEGER IN,IN1,I1,J
      COMMON /TDMARR/A,B,C,D
      INTEGER CAP_LY,CAP_HY
      COMMON /CAP_RANGE/CAP_BLTNEW,CAP_BHTNEW
      INTEGER CAP_BLTNEW,CAP_BHTNEW
      INTEGER CAP2_LA,CAP2_HA
      COMMON /CAP2_RANGE/CAP2_BLTNEW,CAP2_BHTNEW
      INTEGER CAP2_BLTNEW,CAP2_BHTNEW
      COMMON /CAP_GLOBALVARS/CAP_J65SOLVER
      INTEGER CAP_J65SOLVER
      IF ((J.LE.CAP_HY).AND.(J.GE.CAP_LY)) THEN
        IF ((2.LE.CAP2_HA).AND.(2.GE.CAP2_LA)) THEN
          B(2)=B(2)-A(2)*Y(1,J)
        ENDIF
        IF ((IN1.LE.CAP2_HA).AND.(IN1.GE.CAP2_LA)) THEN
          B(IN1)=B(IN1)-C(IN1)*Y(IN,J)
        ENDIF
C
C        FORWARD ELIMINATION
C
        CALL CAP_RECEIVE(D(CAP2_BLTNEW-1),1,2,CAP_UP2)
        CALL CAP_RECEIVE(B(CAP2_BLTNEW-1),1,2,CAP_UP2)
          DO 10  I=MAX(3,CAP2_LA),MIN(IN1,CAP2_HA),1
          I1=I-1
          M1=A(I)/D(I1)
          B(I)=B(I)-M1*B(I1)
          D(I)=D(I)-M1*C(I1)
   10     CONTINUE
        CALL CAP_SEND(D(CAP2_BHTNEW),1,2,CAP_DOWN2)
        CALL CAP_SEND(B(CAP2_BHTNEW),1,2,CAP_DOWN2)
C
C        BACK SUBSTITUTION
C
        IF ((IN1.LE.CAP2_HA).AND.(IN1.GE.CAP2_LA)) THEN
          Y(IN1,J)=B(IN1)/D(IN1)
        ENDIF
        CALL CAP_RECEIVE(Y(CAP2_BHTNEW+1,CAP_J65SOLVER),1,2,CAP_DOWN2)
          DO 20  I=MIN(IN-2,-IN1+CAP2_HA+IN-1),MAX(2,-IN1+CAP2_LA+IN-1),
     +    -1
          Y(I,J)=(B(I)-Y(I+1,J)*C(I))/D(I)
   20     CONTINUE
        CALL CAP_SEND(Y(CAP2_BLTNEW,CAP_J65SOLVER),1,2,CAP_UP2)
      ENDIF
      RETURN
      END


      SUBROUTINE RESIDUAL(LSWEEP,TNEW,IN1,RESIDJ,J,JNM1,TOP,BOT,
     +CAP_LTNEW,CAP_HTNEW,CAP2_LLSWEEP,CAP2_HLSWEEP)
      INTEGER CAP_UP2,CAP_DOWN2
      PARAMETER (CAP_UP2=-3,CAP_DOWN2=-4)
      INTEGER CAP_LEFT,CAP_RIGHT
      PARAMETER (CAP_LEFT=-1,CAP_RIGHT=-2)
C
C     THIS ROUTINE CALCULATES THE RESIDUAL FOR A GIVEN ROW.
C     FOR ALL I ROWS
C
      REAL LSWEEP(500),TNEW(500,500),RESIDJ,NORM
      INTEGER IN1,J
      INTEGER CAP_LTNEW,CAP_HTNEW
      COMMON /CAP_RANGE/CAP_BLTNEW,CAP_BHTNEW
      INTEGER CAP_BLTNEW,CAP_BHTNEW
      INTEGER CAP2_LLSWEEP,CAP2_HLSWEEP
      COMMON /CAP2_RANGE/CAP2_BLTNEW,CAP2_BHTNEW
      INTEGER CAP2_BLTNEW,CAP2_BHTNEW
      EXTERNAL CAP_RADD
      REAL CAP_BOT,CAP_TOP,CAP_RADD
      COMMON /CAP_GLOBALVARS/CAP_J65SOLVER
      INTEGER CAP_J65SOLVER
      IF ((J.LE.CAP_HTNEW).AND.(J.GE.CAP_LTNEW)) THEN
        CAP_TOP=0
        CAP_BOT=0
          DO 10  I=MAX(2,CAP2_LLSWEEP),MIN(IN1,CAP2_HLSWEEP),1
          CAP_TOP=CAP_TOP+(ABS(LSWEEP(I)-TNEW(I,J))+1.E-20)**2
          CAP_BOT=CAP_BOT+TNEW(I,J)*TNEW(I,J)
   10     CONTINUE
        CALL CAP_DCOMMUTATIVE(CAP_BOT,2,CAP_RADD,CAP_UP2)
        BOT=BOT+CAP_BOT
        CALL CAP_DCOMMUTATIVE(CAP_TOP,2,CAP_RADD,CAP_UP2)
        TOP=TOP+CAP_TOP
      ENDIF
      RETURN
      END


      REAL FUNCTION SECOND()
      INTEGER CAP_UP2,CAP_DOWN2
```

```
      PARAMETER (CAP_UP2=-3,CAP_DOWN2=-4)
      INTEGER CAP_LEFT,CAP_RIGHT
      PARAMETER (CAP_LEFT=-1,CAP_RIGHT=-2)
C
C     CREATED BY CI
C
      COMMON /CAP_RANGE/CAP_BLTNEW,CAP_BHTNEW
      INTEGER CAP_BLTNEW,CAP_BHTNEW
      COMMON /CAP2_RANGE/CAP2_BLTNEW,CAP2_BHTNEW
      INTEGER CAP2_BLTNEW,CAP2_BHTNEW
      COMMON /CAP_GLOBALVARS/CAP_J65SOLVER
      INTEGER CAP_J65SOLVER
      SECOND=0
      RETURN
      END
```

Bibliography

---

1 http://hurricanes.noaa.gov/prepare

2 http://www.greenpeace.ceusa.org/features/floydtext.htm

3 http://www.pmel.noaa.gov/tao/elnino/nino-home.html

4 http://www.ctbto.org

5 C. Bailey, P. Chow, Y. Fryer, M. Cross, and K. Pericleous. Multiphysics Modelling Of The Metals Casting Processes. In Proceedings of Royal Society of London A, 452:459--486, 1996.

6 M.J. Flynn. Some Computer Organizations And Their Effectiveness. IEEE Transactions on Computers, C-21:948-960, 1972.

7 K. McManus. A Strategy For Mapping Unstructured Mesh Computational Mechanics Programs Onto Distributed Memory Parallel Architectures. PhD Thesis, Computing and Mathematical Science, University of Greenwich, 1996.

8 http://www.nas.nasa.gov/Groups/Tools/Projects/LCM/

9 http://www.openmp.org

10 M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S-W. Liao, E. Bugnion, M.S. Lam. Maximizing Multiprocessor Performance With The SUIF Compiler. IEEE Computer. December 1996.

11 M. Lam. Locality Optimisations For Parallel Machines. Parallel Processing: CONPAR 94, 1994.

12 http://polaris.cs.uiuc.edu/polaris/polaris.html

13 H.P.F. Forum, High Performance FORTRAN Language Specification, Version 2.0, Rice University, Houston, Texas (1996).

14 M. Frumkin, M. Hribar, H. Jin, A. Waheed and J. Yan. A Comparison Of Automatic Parallelization Tools/Compilers On The SGI Origin 2000. In Proceedings from SC98, Orlando, Florida, November 8-13, 1998.

15 http://www-fp.mcs.anl.gov/petsc

16 http://www.nag.co.uk

17 FORGE90, Applied Parallel Research, Placerville, California 95667, USA.

18 B.M. Chapman, S. Benkner, R. Blasko, P. Brezany, M. Egg, T. Fahringer, H.M. Gerndt, J. Hulman, P. Kutschera, H. Moritsch, A. Schwald, V. Sipkova and H.P. Zima. Vienna Fortran Compilation System Version 1.2. User's Guide. 1996.

19 V.S. Adve, A. Carle, E. Granston, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, J. Mellor-Crummey, S. Warren and C-W. Tseng. Requirements For Data-Parallel Programming Environments. IEEE Parallel and Distributed Technology: Systems and Applications, 2(3):48--58, Fall 1994.

20 P. Banerjee, J.A. Chandy, M. Gupta, E.W. Hodges IV, J.G. Holm, A. Lain, D.J. Palermo, S. Ramaswamy and E. Su. An Overview Of The PARADIGM Compiler For Distributed-Memory Multicomputers. IEEE Computer Volume 28, Number 10, 1995.

21 K.D. Cooper, M.W. Hall, R.T. Hood, K. Kennedy, K.S. McKinley, J.M. Mellor-Crummey, L. Torczon and S.K. Warren. The ParaScope Parallel Programming Environment. In Proceedings of the IEEE, 81(2):244--263, February 1993.

22 http://www.cs.ucsd.edu/groups/hpcl/scg/kelp/

23 C.S. Ierotheou, S.P. Johnson, M. Cross and P.F. Leggett. Computer Aided Parallelisation Tools (CAPTools) - Conceptual Overview and Performance On The Parallelisation of Structured Mesh Codes. Parallel Computing, Vol. 22, pp.163:195, 1996.

24 S.P. Johnson, M. Cross and M.G. Everett. Exploitation Of Symbolic Information In Interprocedural Dependence Analysis. Parallel Computing, 22:197--226, 1996.

25 S.P. Johnson, C.S. Ierotheou, and M. Cross. Automatic Parallel Code Generation For Message Passing On Distributed Memory Systems. Parallel Computing, 22(2):227--258, 1996.

26 P.F. Leggett, A.T.J. Marsh, S.P. Johnson, and M. Cross. Integrating User Knowledge With Information From Parallelisation Tools To Facilitate The Automatic Generation Of Efficient Parallel FORTRAN Code. Parallel Computing, 22:259--288, 1996.

27 F. Darema,-Rogers, V.A. Norton and G.F. Pfister. Using A Single-Program-Multiple-Data Computational Model For Parallel Execution Of Scientific Applications. Technical Report RC11552, IBM T.J. Watson Research Center, November 1985.

28 E.W. Evans, S.P. Johnson, P.F. Leggett and M. Cross. Automatic And Effective Multi-Dimensional Parallelisation Of Structured Mesh Based Codes. Parallel Computing 26, pp 677-703, 2000.

29 E.W. Evans. Strategies And Tools For The Exploitation Of Massively Parallel Computer Systems. PhD Thesis. University of Greenwich, 2000.

30 B. Maerten, D. Roose, A. Basermann, J. Fingberg and G. Lonsdale. DRAMA: A Library For Parallel Dynamic Load Balancing Of Finite Element Applications. In Ninth SIAM Conference on Parallel Processing for Scientific Computing, 1999.

31 R. Volpe and P. Khosla. A Theoretical And Experimental Investigation Of Impact Control For Manipulators. International Journal of Robotics Research, 12(4):351--365, August 1993.

32 http://www.met-office.gov.uk

33 http://www.lanl.gov/asci

34 R.K. Brunner and L. Kalé. Adapting To Load On Workstation Clusters. The Seventh Symposium on the Frontiers of Massively Parallel Computation.

35 H. Nishikawa and P. Steenkiste. A General Architecture For Load Balancing In A Distributed-Memory Environment. International Conference on Distributed Computing, May 1993.

36 R. Lüling and B. Monien. A Dynamic Distributed Load Balancing Algorithm With Provable Good Performance. In Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures, pages 164--172, June 1993.

37 A. Corradi, L. Leonardi, F. Zambonelli. On The Effectiveness Of Different Diffusive Load Balancing Policies In Dynamic Applications. Conference on High-Performance Computing and Networking (HPCN-98), Lecture Notes in Computer Science, No. 1401, Springer-Verlag (D), April 1998.

38 M.A.L. Kalyani, R. Wait and D.N. Ranasinghe. Load Balancing For Distributed Memory Multiprocessors. ITTC2000, Colombo, January 2001.

39 T.D. Nguyen, R. Vaswani, and J. Zahorjan. Using Runtime Measured Workload Characteristics In Parallel Processor Scheduling. In Proceedings of IPPS '96 Workshop on Job Scheduling Strategies for Parallel Processing, pages 93--104, April 1996.

40 H. Shan, J.P. Singh, L. Oliker and R. Biswas. A Comparison Of Three Programming Models For Adaptive Applications On The Origin2000. In Proceedings for SC00, Dallas, TX, 2000.

41 K. Mehrotra, S. Ranka, and J-C. Wang. A Probabilistic Analysis Of A Locality Maintaining Load Balancing Algorithm. In Proceedings of the 7th International Parallel Processing Symposium, pp. 369-373, April 1993.

42 Y.F. Hu and R.J. Blake. An Improved Diffusion Algorithm for Dynamic Load Balancing. Parallel Computing, 25(4):417--444, 1999.

43 Y.F. Hu, R.J. Blake and D.R. Emerson. An Optimal Migration Algorithm For Dynamic Load Balancing. Concurrency: Practice & Experience, 10(6):467--483, 1998.

44 S. Krishnan and L.V. Kalé. Automating Runtime Optimizations For Load Balancing In Irregular Problems. In Proceedings of the Conference on Parallel and Distributed Processing Technology and Applications, San Jose, August 1996.

45 L. Oliker and R. Biswas. Efficient Load Balancing And Data Remapping For Adaptive Grid Calculations. In Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures, 1997.

46 R. Biswas, L. Oliker and A. Sohn. Global Load Balancing With Parallel Mesh Adaption On Distributed-Memory Systems. In Proceedings for SC96, available at http://www.supercomp.org/sc96/proceedings/.

47 C.A. Scheurer, H.K. Scheurer and P.G. Kropf. Load Balancing Driven Process Migration. PMT-Report, University of Berne, June 1995.

48 R. Biswas, S.K. Das, D. Harvey and L. Oliker. Portable Parallel Programming For The Dynamic Load Balancing Of Unstructured Grid Applications. In the 13th International Parallel Processing Symposium, 1999.

49 J. Watts and S. Taylor. A Practical Approach To Dynamic Load Balancing. IEEE Transactions on Parallel and Distributed Systems, 9(3):235--248, 1998.

50 R. Williams. Performance Of Dynamic Load Balancing Algorithms For Unstructured Mesh Calculations. Concurrency: Practice & Experience, 3:457--481, 1991.

51 R. Leland and B. Hendrickson. An Empirical Study Of Static Load Balancing Algorithms. In Proceedings for Scalable High-Performance Computing Conference, IEEE, pp. 682-685, 1994.

52 B. Hendrickson and R. Leland. An Improved Spectral Graph Partitioning Algorithm For Mapping Parallel Computations. SIAM J. Sci. Stat. Comput. Volume 16, 1995.

53 S. Ichikawa and S. Yamashita. Static Load Balancing Of Parallel PDE Solver For Distributed Computing Environment. In Proceedings for ISCA 13th International Conference of Parallel and Distributed Computing Systems, 2000.

54 C. Xu and F. Lau. Load Balancing In Parallel Computers: Theory and Practice. Kluwer Academic Publishers, 1997.

55 G. Cybenko. Dynamic Load Balancing For Distributed Memory Multiprocessors. Journal of Parallel and Distributed Computing, pages 279--301, Volume 7, 1989.

56 M.H. Willebeek-LeMair and A.P. Reeves. Strategies For Dynamic Load Balancing On Highly Parallel Computers. IEEE Transactions on Parallel and Distributed Systems, 4(9):979--993, 1993.

57 J. Xu and K. Hwang. Heuristic Methods For Dynamic Load Balancing In A Message-Passing Multicomputer. Journal of Parallel and Distributed Computing, 18:1--13, 1993.

58 M. Hamdi and C-K. Lee. Dynamic Load Balancing Of Data Parallel Applications On A Distributed Network. Parallel Computing, 22:1477--1492, 1997.

59 J.N. Rodrigues, S.P. Johnson, C. Walshaw and M. Cross. An Automatable Generic Strategy For Dynamic Load Balancing In A Parallel Structured Mesh CFD Code. Parallel Computational Fluid Dynamics: Towards Teraflops, Optimization and Novel Formulations, pp. 345-354, edited by D. Keyes et al. Also in Proceedings for PCFD'99, Williamsburg, 1999.

60 J.N. Rodrigues, S.P. Johnson, C. Walshaw and M. Cross. Automatic Implementation Of Dynamic Load Balancing Strategies For Structured Computational Mechanics Codes. In B.H.V. Topping, editor, Developments In Computational Mechanics with High Performance Computing, pp. 41-47, Civil Comp Press. Also in Proceedings for Parallel & Distributed Computing For Computational Mechanics, Weimar, 1999.

61 C. Walshaw, M. Cross, S. Johnson, and M. Everett. JOSTLE: Partitioning Of Unstructured Meshes For Massively Parallel Machines. Parallel Computational Fluid Dynamics: New Algorithms and Applications, pp. 273-280, Elsevier, Amsterdam, 1995.

62 C. Walshaw. A Parallelisable Algorithm For Optimising Unstructured Mesh Partitions. Technical Report P95/IM/03, School of Computing and Mathematical Science, University of Greenwich. January 1995.

63 C. Walshaw, M. Cross, M.G. Everett, S. Johnson and K. McManus. Partitioning And Mapping Of Unstructured Meshes To Parallel Machine Topologies. International Journal for Supercomputing Applications, 1995.

64 K. McManus, C. Walshaw, M. Cross, P. Leggert and S. Johnson. Evaluation Of The JOSTLE Mesh Partitioning Code For Practical Multiphysics Applications. In Proceedings for PCFD'95, 1995.

65 http://www.gre.ac.uk/~wc06/jostle

66 G. Karypis and V. Kumar. Multilevel K-way Partitioning Scheme For Irregular Graphs. Journal of Parallel and Distributed Computing, pages 96-129, Volume 48 Number 1, 1998.

67 http://www-users.cs.umn.edu/~karypis/metis/

68 M. Cermele, M. Colajanni and S. Tucci. Adaptive Load Balancing Of Distributed SPMD Computations: A Transparent Approach. Technical Report DISP-RR-97.02, Dipartimento di Informatica, Sistemi e Produzione, Universita' di Roma Tor Vergata, 1997.

69 M. Cermele, M. Colajanni and S. Tucci. Check-Load Interval Analysis For Balancing Distributed SPMD Applications. In Proceedings for the International Conference on Parallel and Distributed Techniques and Applications, Las Vegas, v. 1, pp 432--442, June 1997.

70 O. Krone, M. Raab and B. Hirsbrunner. Load Balancing For Network Based Multi-Threaded Applications. In Proceedings for the 5th European PVM/MPI User's Group Meeting (EuroPVM/MPI'98), Liverpool, 1998.

71 T.W. Clark, R. Hanxleden, J.A. McCammon and L. R. Scott. Parallelizing Molecular Dynamics Using Spatial Decomposition. In Proceedings of the Scalable High Performance Computing Conference, Knoxville, TN, May 1994.

72 J.N.C. Árabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey and P. Stephan. Dome: Parallel Programming In A Heterogeneous Multi-User Environment. CMU-CS95 -137 30786, Carnegie Mellon University, School of Computer Science, April 1995.

73 B. Moon and J. Saltz. Adaptive Runtime Support For Direct Simulation Monte Carlo Methods On Distributed Memory Architectures. In Scalable High-Performance Computing Conference, pages 176-183, May 1994.

74 D. Nicol and G. Ciardo. Automated Parallelization Of Discrete State-Space Generation. Journal of Parallel and Distributed Computing, 47(2):153-167, December 1997.

75 D.M. Nicol and J.H. Saltz. Dynamic Remapping Of Parallel Compuations With Varying Resource Demands. IEEE Transactions on Computers, 37(9):1073-1087, September 1988.

76 C.Xu, F. Lau and R. Diekmann. Decentralized Remapping Of Data Parallel Applications In Distributed Memory Multiprocessors. Concurrency: Practice and Experience, pages 1351-1376, December 1997.

77 M. Cermele, M. Colajanni and G. Necci. Dynamic Load Balancing Of Distributed SPMD Computations With Explicit Message-Passing. In Proceedings for IEEE Heterogeneous Computing Workshop, Geneva, pp. 2-13, Apr 1997.

78 M.J. Zaki, W. Li and S. Parthasarthy. Customized Dynamic Load Balancing For A Network Of Workstations. Journal of Parallel and Distributed Computing, 43:156--162, 1997.

79 L.V. Kalé, M. Bhandarkar and R. Brunner. Load Balancing In Parallel Molecular Dynamics. In the Fifth International Symposium on Solving Irregularly Structured Problems in Parallel, 1998.

80 J. Garner, R.J. Allen, D.R. Emerson and R.J. Blake. Practical Dynamic Load Balancing For Multi-Dimensional Domain Decomposition in CFD. In Proceedings for the 8th ACM International Conference on Supercomputing, Manchester, 1994.

81 B.S. Siegell and P. Steenkiste. Automatic Generation Of Parallel Programs With Dynamic Load Balancing. In Proceedings for the Third International Symposium on High-Performance Distributed Computing, August 1994.

82 M. Colajanni and M. Cermele. Dame: An Environment For Preserving Efficiency Of Data Parallel Computations On Distributed Systems. IEEE Concurrency, Volume 5, n. 1, pp. 41--55, Jan.-Mar. 1997.

83 http//:traianus.ce.uniroma2.it/~dame

84 C. Baillie, J. Michalakes and R. Skålin. Regional Weather Modeling On Parallel Computers. Parallel Computing, 23, pp. 2135--2142, 1997

85 R.B.P. Burrows. Dynamic Load Balancing. PhD Thesis, Oxford, 1997.

86 N-T. Fong, C-Z. Xu and L.Y. Wang. Optimal Periodic Remapping Of Bulk Synchronous Computations On Multiprogrammed Distributed Systems. IPDPS 2000.

87 A. Arulananthan, S.P. Johnson, K. McManus, C. Walshaw and M. Cross. A Generic Strategy For Dynamic Load Balancing Of Distributed Memory Parallel Computational Mechanics Using Unstructured Meshes. In Proceedings for Parallel CFD '97, Machester, 1997.

88 http://www.nas.nasa.gov/Software/NPB/

89 M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Scharzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum and J. Martin. The PERFECT Club Benchmarks: Effective Performance Evaluation Of Supercomputers. The International Journal of Supercomputer Applications, 3(3):5-40, 1989.

90 http://www.mth.uea.ac.uk/ocean/SEA/

91 L. De Rose, K. Gallivan and E. Gallopoulos. 3-D Land Avoidance And Load Balancing In Regional Ocean Simulation. ICPP, Vol. 2 (1996).

92 J. Drake, I. Foster, J. Michalakes, B. Toonen and P. Worley. Design And Performance Of A Scalable Parallel Community Climate Model. Parallel Computing 21, pp. 1571--1591, 1995.

93 K. Devine, B. Hendrickson, E. Boman, M. St. John and C. Vaughan. Design of Dynamic Load-Balancing Tools for Parallel Applications. Parallel Computing, 21(10):1571--1591, 1995.

94 R.W. Ford and J.M. Bull. Dynamic Load Balancing In The UKMO's Unified Model. Presented at the Workshop on Dynamic Load Balancing on MPP Systems Daresbury U.K., November 1995.

95 B. Hendrickson and K. Devine. Dynamic Load Balancing In Computational Mechanics. Computer Methods in Applied Mechanics and Engineering, 184:485-500, 2000.

96 Z. Lan, V. Taylor and G. Bryan. Dynamic Load Balancing For Structured Adaptive Mesh Refinement Applications. In Proceedings for the 30th International Conference on Parallel Processing, Valencia, Spain, 2001.

97 S.J. Plimpton, D.B. Seidel, M.F. Pasik and R.S. Coats. Novel Load-Balancing Techniques For An Electromagnetic Particle-In-Cell Code. Technical Report SAND2000-0035, Sandia National Laboratories, Albuquerque, NM, 2000.

98 P. Burton, R. Oxford and D. Salmond. Investigation And Implementation Of A Number Of Different Load Balancing Strategies In The UK Met. Office's Unified Model. In Proceedings for the 4th European SGI/Cray MPP Workshop, Institute for Plasma Physics, Garching/Munich, Germany, September 1998.

99 http://www.specbench.org/

100 S.P. Johnson, C. Ierotheou and M. Cross. Computer Aided Parallelisation Of Unstructured Mesh Codes. In Proceedings for PDPTA 1997, Volume 1, pages 344--353, July 1997.

101 C. Ierotheou, S.P. Johnson, K. McManus, P.F. Leggett and M. Cross. Semiautomatic Parallelisation Of Unstructured Mesh Codes. Parallel CFD, Holland, May 1997.

102 C. Walshaw and M. Berzins. Dynamic Load-Balancing For PDE Solvers On Adaptive Unstructured Meshes. Concurrency: Practice and Experience, Volume 7 (1), 17-28, 1995.

103 C. Walshaw, M. Cross and M. Everett. Parallel Dynamic Graph Partitioning For Adaptive Unstructured Meshes. Journal of Parallel and Distributed Computing, 47, 102-108, 1997.

104 C. Walshaw, M. Cross and K. McManus. Multiphase Mesh Partitioning. Applied Mathematical Modelling, 25, 123-140, 2000.

105 C. Walshaw and M. Cross. Parallel Optimisation Algorithms For Multilevel Mesh Partitioning. Parallel Computing, 26, 1635-1660, 2000.

106 P. Chow. A Control Volume Unstructured Mesh Procedure For Convection-Diffusion Solidification Processes. PhD Thesis, University of Greenwich, 1993.

107 Y.D. Fryer, C. Bailey, M. Cross and C-H. Lai. A Control Volume Procedure For Solving The Elastic Stress-Strain Equations On An Unstructured Mesh. Applied Mathematical Modelling, 15:639--645, 1991.

108 M. Cross, C. Bailey, P. Chow, K. Pericleous and Y.D. Fryer. Towards An Integrated Control Volume Unstructured Mesh Code For The Simulation Of All The Macroscopic Processes Involved In Shape Casting. Numerical Methods in Industrial Forming Processes, (NUMIFORM 92), pages 787-792, Balkema. 1992.

109 B.W. Jones, K. McManus, M. Cross, M.G. Everett and S. Johnson. Parallel Unstructured Mesh CFD Codes: A Role For Recursive Clustering Techniques In Mesh Decomposition. Parallel Computational Fluid Dynamics: New Trends and Advances, Elsevier Science B.V. Pages 207--214, 1995.

110 http://www.gridforum.org/ and http://www.globus.org/research/papers.html

111 S.P. Johnson. Mapping Numerical Software Onto Distributed Memory Parallel Systems. PhD Thesis, University of Greenwich. 1992.

112 P.F. Leggett, S.P. Johnson and M. Cross. CAPLib – A 'Thin Layer' Message Passing Library To Support Computational Mechanics Codes On Distributed Memory Parallel Systems. Advances in Engineering Software 32, pp. 61-83, Elsevier, 2001.

113 Computer Aided Parallelisation Tools (CAPTools) User Manual Release 1.0, University of Greenwich. 1994.

114 A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Nanchek and V. Sunderam. PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing. Scientific and Engineering Series. MIT Press, 1994. ISBN: 0-262-57108-0.

115 Message Passing Interface Forum. The MPI Message Passing Interface Standard, Technical Report, University of Tennessee, Knoxville. 1994.

116 Cray Research Inc. SHMEM Technical Note for C. SG-2516 2.3. October 1994.

117 R. Bornat. Understanding And Writing Compilers. Macmillan Educational Limited, London. 1979.

118 A.V. Aho, R. Sethi and J.D. Ullman. Principals Of Compiler Design, Addison-Wesley, Reading, MA, 1986.

119 J. Ferrante, K.J. Ottenstein and J.D. Warren. The Program Dependence Graph And Its Use In Optimisation. ACM Transactions On Programming Languages And Systems, 9:319--349, 1987.

120 D.J. Kuck. The Structure Of Computers And Computations, Vol 1, Wiley, New York, 1978.

121 J.R. Allen and K. Kennedy. Automatic Translation Of Fortran Programs To Vector Form. ACM Transactions on Programming Languages and Systems, 9(4):491--542. 1987.

122 H.P. Zima and B. Chapman. Supercompilers For Parallel And Vector Computers. ACM Press Frontier Series, Addison-Wesley, New York, 1990.

123 M. Haghighat and C. Polychronopoulos. Symbolic Program Analysis And Optimisation For Parallelising Compilers. In Conference Record of the 5th Workshop on Languages and Compilers for Parallel Computing, Yale University, Department of Computer Science, 1992.

124 U. Banerjee. Speedup Of Ordinary Programs. PhD Thesis, University of Illinois at Urbana Champaign. 1979.

125 U. Banerjee. Dependence Analysis For Supercomputing. Kluwer Academic Publishers, Boston, MA, 1988.

126 W. Pugh. A Practical Algorithm For Exact Array Dependence Analysis. Communications of the ACM, 35(8):27--47, 1992.

127 R.A. Frost. Introduction To Knowledge Based Systems. Collins Professional and Technical Series, London. 1986.

128 C.H.Q. Ding. Evaluations Of HPF For Practical Scientific Algorithms. In Springer-Verlag Lecture Notes on Computer Science, vol. 1401, 1998, Ed. by P.M.A. Sloot, pp:223-232.

129 R. Thakur, A. Choudhary and G. Fox. Runtime Array Redistribution In HPF Programs. In Proceedings for Scalable High Performance Computing Conference, pp. 309--316, May 1994.

130 L.M. Delves. Porting Industrial Codes To MPP Systems Using HPF. High Performance Computing, pp. 103-112. Edited by R.J. Allan et al., Kluwer Academic / Plenum Publishers, New York, 1999.